计算机组成原理

计算机组成原理课程组
（刘旭东、高小鹏、肖利民、牛建伟、栾钟治）

第二部分：组合逻辑

Verilog HDL概述

- 硬件描述语言（Hradware Description Language）是一种用形式化方法（即文本形式）来描述和设计数字电路和数字系统的高级模块化语言。它是设计人员和EDA工具之间的一个桥梁，主要用于编写设计文件，在EDA工具中建立电路模型；也用来编写测试文件进行仿真。

用HDL描述设计 —EDA工具→ 综合、仿真 → 目标文件 → 编程下载

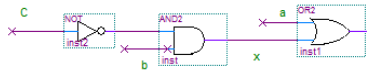
- HDL发展至今已有近三十年的历史，到20世纪80年代，已出现了数十种硬件描述语言。80年代后期，HDL向着标准化、集成化的方向发展，最终VHDL、Verilog HDL先后成为IEEE标准。
  - VHDL：VHSIC Hardware Description Language（VHSIC——Very High Speed Integrated Circuits），甚高速集成电路的硬件描述语言，来源于美国军方，1987年成为IEEE标准。目前标准化程度最高的一种HDL。

Verilog HDL概述

- Verilog HDL可以用来进行数字电路的建模、仿真验证、时序分析、逻辑综合。
- Verilog HDL抽象级别：系统级，算法级，RTL级，门级，开关级
- Verilog HDL具有行为描述和结构描述功能。行为描述包括系统级、算法级和RTL级3种抽象级别；结构描述包括包括门级和开关级2种抽象级别。
- 语法结构上的主要特点
  - 形式化地表示电路的行为和结构；
  - 借用C语言的结构和语句；
  - 可在多个层次上对所设计的系统加以描述，语言对设计规模不加任何限制；
  - 具有混合建模能力：一个设计中的各子模块可用不同级别的抽象模型来描述；
  - 基本逻辑门、开关级结构模型均内置于Verilog HDL语言库中，可直接调用；

Verilog HDL模块的结构

- Verilog的基本设计单元是“模块（module）”，实现一个特定功能
- 一个“与门”、“加法器”、ALU都可以是一个模块
- Verilog模块的结构由在module和endmodule关键词之间的4个主要部分组成：



1. 端口定义
2. I/O说明
3. 信号类型声明
4. 功能描述

```
module block1(a,b,c,d );
    input a, b, c;  /* I/O变量缺省为wire变量*/
    output d;
    wire x;
    assign d = a | x; //组合逻辑功能描述
    assign x = ( b & ~c );
endmodule
```

结构描述

Verilog HDL实例

一、简单的Verilog HDL例子

【例1】8位全加器

模块名(文件名)　端口定义　I/O说明　功能描述

```
module  adder8 ( cout,sum,a,b,cin );
   output  cout;                  // 输出端口声明
   output [7:0] sum;
   input [7:0] a,b;               // 输入端口声明
   input   cin;
   assign {cout,sum}=a+b+cin;
endmodule
```

- 整个程序嵌套在module和endmodule声明语句中。
- 每条语句相对module和endmodule最好缩进2格或4格！
- // …… 表示注释部分，一般只占据一行。对编译不起作用！
- []: 数组/总线定义
- {}: 位拼接运算符

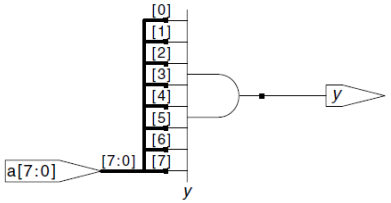
单行注释符

Verilog HDL实例

【例2】8位与门

```
module and8 (input [7:0] a, output y);
   assign y  &a;  // &a is much easier to write than
   // assign y= a[7] & a[6] & a[5] & a[4] & a[3] & a[2] & a[1] & a[0];
endmodule
```

- &：缩位操作符

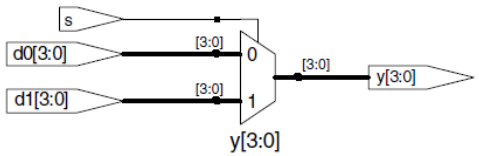


Verilog HDL实例

【例3】多路选择器

```
module mux2 (input [3:0] d0, d1,
  input s,   output [3:0] y);
  assign y = s ? d1 : d0;
endmodule
```

- ?：条件运算符

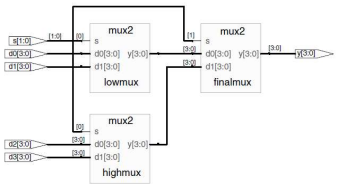


Verilog HDL实例

【例4】 多路选择器

```
module mux4 (input [3:0] d0, d1, d2, d3, input [1:0] s, output [3:0] y);
  wire [3:0] low, high;
  mux2 lowmux (d0, d1, s[0], low);
  mux2 highmux (d2, d3, s[0], high);
  mux2 finalmux (low, high, s[1], y);
Endmodule

module mux2 (input [3:0] d0, d1, input s, output [3:0] y);
  tristate t0 (d0, ~s, y);
  tristate t1 (d1, s, y);
endmodule
```



Verilog HDL模块的结构

1、模块端口定义

- 模块端口定义用来声明设计电路模块的输入输出端口，端口定义格式：
  module 模块名(端口1, 端口2, 端口3, …);
- 在端口定义的圆括弧中，是设计电路模块与外界联系的全部输入输出端口信号或引脚，它是设计实体对外的一个通信界面，是外界可以看到的部分（不包含电源和接地端），多个端口名之间用“,”分隔。

2、 I/O说明

- 模块的I/O说明用来声明模块端口定义中各端口数据流动方向，包括输入（input）、输出（output）和双向（inout）。
  input 端口1, 端口2, 端口3, …;
  output 端口1, 端口2, 端口3, …;
  inout 端口1, 端口2, 端口3, …;

- 端口定义、I/O说明和程序语句中的标点符号及圆括弧均要求用半角符号书写！

Verilog HDL模块的结构

3、信号类型声明

- 信号类型声明用来说明设计电路的功能描述中，所用的信号的数据类型以及函数声明。
- 信号的数据类型主要有连线（wire）、寄存器（reg）、整型（integer）、实型（real）和时间（time）等类型。

4、功能描述

- 功能描述是Verilog HDL程序设计中最主要的部分，用来描述设计模块的内部结构和模块端口间的逻辑关系，在电路上相当于器件的内部电路结构。
- 功能描述可以用assign语句、元件例化（instantiate）、always块语句等方法来实现，通常把确定这些设计模块描述的方法称为建模。

逻辑功能定义

❖在Verilog 模块中有3种方法可以描述电路的逻辑功能：

① 用assign 语句

数据流描述 — 常用于描述组合逻辑

```
assign x = ( b & ~c );
```

② 用元件例化（instantiate）

```
and myand3( f,a,b,c);
```

结构描述

门元件关键字 — and

例化元件名 — myand3

- ❖ 注1：元件例化即是调用Verilog HDL提供的元件或由某子模块定义的实例元件；
- ❖ 注2：元件例化包括门元件例化和模块元件例化；
- ❖ 注3：例化元件名也可以省略！若不省略，则每个实例元件的名字必须唯一！以避免与其它调用该元件的实例相混淆。

逻辑功能定义（续）

③ 用“always”块语句 **行为描述**

```
always @(posedge clk) // 每当时钟上升沿到来时执行一遍块内语句
  begin
    if(load)      out = data;            // 同步预置数据
    else          out = data + 1 + cin;  // 加1计数
  end
```

- 注1：“always” 块语句常用于描述时序逻辑，也可描述组合逻辑。
- 注2：“always” 块可用多种手段来表达逻辑关系，如用if-else语句或case语句。
- 注3：“always” 块语句与assign语句是并发执行的，assign语句一定要放在“always” 块语句之外！

Verilog HDL语句

- 语句及函数的比较

语句及函数	C语言	Verilog HDL
函数	无参函数，有参函数	function块语句
赋值语句	赋值变量 = 表达式;	阻塞赋值，非阻塞赋值
条件语句	if-else	if-else
条件语句	switch	case
循环语句	for	for
循环语句	while	while
中止语句	break	break
宏定义语句	define（以符号#开头）	define（以符号`开头）
格式输出函数	printf	printf

运算符的比较

C语言	Verilog HDL	功能	C语言	Verilog HDL	功能
+	+	加	<=	<=	小于等于
-	-	减	==	==	等于
*	*	乘	!=	!=	不等于
/	/	除	~	~	按位取反
%	%	取模	&	&	按位与
!	!	逻辑非	|	|	按位或
&&	&&	逻辑与	^	^	按位异或
||	||	逻辑或	<<	<<	左移
>	>	大于	>>	>>	右移
<	<	小于	?:	?:	等同于if-else
>=	>=	大于等于			

Verilog HDL与C语言的运算符几乎完全相同！

Verilog HDL模块的结构

- Verilog HDL程序的三种描述方式

结构（Structural）描述
- 对设计电路的结构进行描述，即描述设计电路使用的元件及这些元件之间的连接关系
- 属于低层次的描述方法，包括门级和开关级2种抽象级别

行为（Behavioural）描述
- 对设计电路的逻辑功能的描述，并不用关心设计电路使用哪些元件以及这些元件之间的连接关系
- 属于高层次的描述方法，包括系统级、算法级和寄存器传输级等3种抽象级别

数据流（Data Flow）描述
- 采用持续赋值语句

Verilog HDL模块

❖Verilog HDL模块的模板（仅考虑用于逻辑综合的程序）

```
module <顶层模块名> (< 输入输出端口列表>) ;
  output 输出端口列表;
  input 输入端口列表;
//（1）使用assign语句定义逻辑功能
  wire <结果信号名> ;
  assign <结果信号名> = 表达式 ;
//（2）使用always块定义逻辑功能
  always @(<敏感信号表达式>)
    begin
      //过程赋值语句
      //if语句
      //case语句
      //while,repeat,for循环语句
      //task,function调用
    end
```


Verilog HDL模块的模板

❖ Verilog HDL模块的模板（续）

```
//（3）元件例化
  < module_name > < instance_name > (<port_list>);  // 模块元件例化
  <gate_type_keyword> < instance_name > (<port_list>); // 门元件例化
endmodule
```

例化元件名也可以省略！

Verilog HDL的词法

❖ Verilog HDL源程序由空白符分隔的词法符号流组成。

❖ 词法符号包括空白符、注释、操作符（运算符）、常数、字符串、标识符及关键字。

一、空白符和注释

❖Verilog HDL的空白符包括空格、Tab、换行和换页符号。空白符如果不是出现在字符串中，编译源程序时将被忽略。

❖注释用来帮助读者理解程序，编译源程序时将被忽略。注释分为行注释和块注释两种方式。

- 行注释用符号//（两个斜杠）开始，注释到本行结束。
- 块注释用/*开始，用*/结束。块注释可以跨越多行，但它们不能嵌套。

二、常数

❖ 常数包括整数、x（未知）和z（高阻）值、负数、实数

❖ 整数的4种进制表示形式：

- 二进制整数（b或B）；
- 十进制整数（d或D）；
- 十六进制整数（h或H）；
- 八进制整数（o或O）。

建议最好写明位宽和进制——清楚，不易出错！

整数的3种表达方式	说 明	举 例
<位宽> '<进制符号> <数字>	完整的表达方式	8'b11000101或8 'hc5
<进制符号> <数字>	缺省位宽，则位宽由机器系统决定，至少32位	hc5
<数字>	缺省进制为十进制，位宽默认为32位	197

❖ 这里位宽指对应二进制数的宽度。

❖ 整数型常量是可以综合的，而实数型和字符串型常量都是不可综合的

x和z值

❖x和z值
- x表示不定值，z表示高阻值；
- x和z代表的二进制位数取决于所用的进制

“？”是z的另一种表示符号，建议在case语句中使用？表示高阻态z

【例2.17 】casez (select)
```
4'b???1: out = a;
4'b??1?: out = b;
4'b?1??: out = c;
4'b1???: out = d;
endcase
```

❖ 负数
- 在位宽前加一个负号，即表示负数，负数通常表示为该负数的二进制补码
- 如：-8'd5　//-5的补码，= 8'b11111011

实数（Real）

❖实数的两种表示法
- 十进制表示法
  - 2.0，5.678，0.1　　//合法
  - 2.　　//非法，小数点两侧都必须有数字
- 科学计数法
  - 43_5.1e2　　// 等于$435.1\times10^2=43510$
  - 5E-4　　//等于$5\times10^{-4}=0.0005$，e与E相同

❖实数通过四舍五入被转换为最相近的整数

【例】42.446，42.45　//若转换为整数都是42
92.5，92.699　//若转换为整数都是93
-15.62，-25.26　//若转换为整数分别为-16，-25

- 下划线“_”可随意用在整数或实数的数字中间，以提高可读性；但数字的第1个字符不能是下划线，也不能用在位宽和进制处。

三、字符串

❖字符串是用双引号括起来的可打印字符序列，不能多行书写。

❖作用：在仿真时显示一些相关信息，或者指定显示的格式
- 例：”INTERNAL ERROR”， ”this is an example for Verilog HDL”

❖字符串能够用在系统任务（如$display、$monitor）中作为变量，字符串的值可像数值一样存储在寄存器中，也可以像对数字一样对字符串进行赋值、比较和拼接操作

【例】$display($time,,,”a=%h b=%h c=%h”,a,b,c);
// 显示当前仿真时间，空3格后显示a=xx b=xx c=xx

❖ 字符串属于reg型变量，宽度为字符串中字符的个数乘以8。

【例】
```
reg[8*12:1] stringvar;
initial
  begin
    stringvar = “Hello world!”;
  end
```


四、标识符

❖任何用Verilog HDL语言描述的对象都通过其名字来识别，这个名字被称为标识符。标识符可由字母、数字、下划线和$符号构成。

❖如源文件名、模块名、端口名、变量名、常量名、实例名等。

❖定义标识符时应遵循如下规则

① 首字符必须是字母或下划线，不能是数字或$符号！
② 字符数不能多于1024个。
③ 大小写字母是不同的。
④ 不要与关键字同名！

❖ 合法的名字：
- A_99_Z
- Reset
- _54MHz_Clock$
- Module

❖ 不合法的名字：
- 123a
- $data
- module
- 7seg.v
- out*　　//不允许包含字符*

五、关键字

- 关键字（保留字）——Verilog HDL事先定义好的确认符，用来组织语言结构；或者用于定义Verilog HDL提供的门元件（如and，not，or，buf）。
- 每个关键字全部用小写字母定义！
  ——如always，assign，begin，case，casex，else，end，endmodule，for，function，if，input，module，output，repeat，table，time，while，wire
- Verilog -1995的关键字有97个，Verilog -2001增加了5个共102个。

六、运算符及表达式

- 运算符也称为操作符，是Verilog HDL预定义的函数符号，这些函数对被操作的对象（即操作数）进行规定的运算，得到一个结果。

- 运算符按功能分为9类：
  - 算术运算符
  - 逻辑运算符
  - 关系运算符
  - 等值运算符
  - 缩减运算符
  - 条件运算符
  - 位运算符
  - 移位运算符
  - 位拼接运算符

- 运算符按操作数的个数分为3类：
  - 单目运算符——带一个操作数
  - 逻辑非！，按位取反~，缩减运算符，移位运算符
  - 双目运算符——带两个操作数
  - 算术、关系、等值运算符，逻辑运算符（除逻辑非外）、位运算符（除按位取反外）
  - 三目运算符——带三个操作数
  - 条件运算符

1、算术运算符

- 双目运算符

算术运算符	功能
+	加
−	减
*	乘
/	除
%	求模

- 进行整数除法运算时，结果值略去小数部分，只取整数部分！
- 求模即是求一个数被另一个数相除后所得的余数。%称为求模（或求余）运算符，要求%两侧均为整型数据；
- 求模运算结果值的符号位取第一个操作数的符号位！
  【例】-11%3　结果为-2
- 进行算术运算时，若某操作数为不定值x，则整个结果也为x。

2、逻辑运算符

- 逻辑运算符把它的操作数当作布尔变量（逻辑1、逻辑0或不定值）：
  - 非零的操作数被认为是真(1'b1)；
  - 零被认为是假(1'b0)；
  - 不确定的操作数如4'bxx00，被认为是不确定的（可能为零，也可能为非零）（记为1'bx）；但4'bxx11被认为是真（记为1'b1，因为它肯定是非零的）。

【例】若A=4'b0000，B=4'b0101
则 !A=1'b1，A&&B=1'b0，A||B=1'b1。

逻辑运算符	功能
&&(双目)	逻辑与
||(双目)	逻辑或
!(单目)	逻辑非

- 如果操作数不止一位，应将操作数作为一个整体来对待！
- 进行逻辑运算后的结果为布尔值（为1或0或x）！

3、位运算符

单目运算符

双目运算符

位运算符	功能
~	按位取反
&	按位与
|	按位或
^	按位异或
^~，~^	按位同或

- 位运算符中的双目运算符要求对两个操作数的相应位逐位进行逻辑运算。位运算其结果与操作数位数相同。
- 两个不同长度的操作数进行位运算时，将自动按右端对齐，位数少的操作数会在高位用0补齐。

【例】若A = 5'b11001，B = 3'b101，
则A & B = （5'b11001）&（5'b00101）= 5'b00001

4、关系运算符

- 双目运算符
- 用来对两个操作数进行比较。

也是非阻塞赋值运算的赋值符号

关系运算符	功能
<	小于
<=	小于或等于
>	大于
>=	大于或等于

- 运算结果为1位的逻辑值1或0或x。关系运算时，若声明的关系为真，则返回值为1；若关系为假，则返回值为0；若某操作数为不定值x，则返回值为x，表示结果是模糊的。
- 所有的关系运算符优先级别相同。
- 关系运算符的优先级低于算术运算符。

【例】a<size - 1　　等同于：a<(size - 1)
size - (1<a)　　不等同于：size-1<a

括号内先运算

算术运算先运算

5、等值运算符

- 双目运算符

Quartus II不支持!

等值运算符	功能
==	等于
!=	不等于
===	全等
!==	不全等

- 运算结果为1位的逻辑值1或0或x。
- 所有的等值运算符优先级别相同。
- ===和!==运算符常用于case表达式的判别，又称为“case等式运算符”。

- 等于运算符(==)和全等运算符(===)的区别:
  - 使用等于运算符时，两个操作数必须逐位相等,结果才为1；若某些位为x或z，则结果为x。
  - 使用全等运算符时，若两个操作数的相应位形式上完全一致（如同是1，或同是0，或同是x，或同是z）,则结果为1；否则为0。

6、缩减（缩位）运算符

- 单目运算符
- 运算法则与位运算符类似，但运算对象只有一个，运算过程不同！

注意缩减运算符和位运算符的区别！

缩减运算符	功能
&	与
~&	与非
|	或
~|	或非
^	异或
^~，~^	同或

位运算符没有

- 对单个操作数进行递推运算,即先将操作数的最低位与第二位进行与、或、与非、或非等运算，再将运算结果与第三位进行相同的运算，依次类推，直至最高位。
- 运算结果缩减为1位二进制数。

【例】reg[3:0] a;
b=|a;　　//等效于 b =((a[0] | a[1]) | a(2)) | a[3]

【例2.18】设A = 8'b11010001，则&A = 0（在与缩减运算中，只有A中的数字全为1时，结果才为1）；|A = 1（在或缩减运算中，只有A中的数字全为0时，结果才为0）。

7、移位运算符

- 单目运算符
- 常用于移位寄存器的设计

移位运算符	功能
>>	右移
<<	左移

- 用法：A>>n　或A<<n

  将操作数右移或左移n位，同时用n个0填补移出的空位。注意操作数的位数不变！

【例】4'b1001>>3 的结果= 4'b0001；　4'b1001>>4的结果= 4'b0000

4'b1001<<1的结果= 4'b0010；　4'b1001<<2的结果= 4'b0100；

1<<6 = 32'b00…01000000

左移的数据会丢失！

右移的数据会丢失！

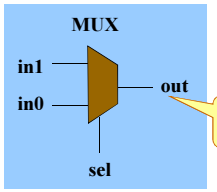
将操作数右移或左移n位，相当于将操作数除以或乘以2^n。

8、条件运算符

- 三目运算符
- 常用于数据选择器的设计
- 条件运算符为？：
- 用法：信号 = 条件？表达式1：表达式2;

当条件为真，信号取表达式1的值；为假，则取表达式2的值。

【例】数据选择器assign out = sel? in1:in0;



sel=1时out=in1；
sel=0时out=in0

9、位拼接运算符

- 位拼接运算符为{ }
- 用于将两个或多个信号的某些位拼接起来，表示一个整体信号。
- 用法：{信号1的某几位, 信号2的某几位, …… , 信号n的某几位}

➢例如在进行全加运算时，可将进位输出与算术和拼接在一起使用。

```
【例2.19】output [3:0] sum;                    //算术和
          output cout;                         //进位输出
          input[3:0] ina,inb;
          input cin;
          assign {cout,sum} = ina + inb +cin；//进位与算术和拼接在一起
```

➢位拼接可以嵌套使用，或用重复法简化书写

```
【例2.20】{3{a,b[3:0]} }
  ={ {a,b[3],b[2],b[1],b[0]}, {a,b[3],b[2],b[1],b[0]}, {a,b[3],b[2],b[1],b[0]}}
```


运算符的优先级

类 别	运 算 符	优先级
逻辑非、按位取反	! ~	高
算术运算符	* / %	
	+ -	
移位运算符	<< >>	
关系运算符	< <= > >=	
等式运算符	== != === !==	
缩减运算符	& ~&	
	^ ^~	
	| ~|	
逻辑运算符	&&	
	||	低
条件运算符	? :	

- 为避免错误，提高程序的可读性，建议使用括号来控制运算的优先级！

【例】（a>b)&&(b>c)

(a= =b)||(x= = y)

(!a)||(a>b)

七、 Verilog HDL数据对象

❖Verilog HDL数据对象是指用来存放各种类型数据的容器，包括常量和变量。

（1）常量

➢常量用来存放一个恒定不变的数据，一般在程序前部定义。用parameter来定义一个标识符，代表一个常量，称为符号常量或parameter常量。

```
parameter 常量名1 = 表达式, 常量名2 = 表达式, …, 常量名n = 表达式;
```

➢parameter是常量定义关键字，常量名是用户定义的标识符，表达式是为常量赋的值。

➢每个赋值语句的右边必须为常数表达式，即只能包含数字或先前定义过的符号常量！

➢parameter常量常用来定义延迟时间和变量宽度。当程序中有多处地方用到相同的常量时，建议用parameter常量来定义—便于修改，有意义

➢parameter常量是本地的，其定义只在本模块内有效。

【例】 parameter addrwidth = 16;

变量

（2）变量

❖在程序运行过程中，其值可以改变的量，称为变量。

❖其数据类型有19种，常用的有3种：

➢网络型（nets type）

➢寄存器型（register type ）

➢数组（memory type）

❖其它数据类型：large型、medium型、scalared型、 small型、time型、tri型、tri0型、tri1型、triand型、trior型、trireg型、vectored型、wand型、wor型等

nets型变量

1. nets型变量

❖ 网络型变量（ nets型变量）是输出值始终随输入的变化而变化的变量。

❖ 一般用来定义电路中的各种物理连线。

❖ 有两种驱动方式：在结构描述中将其连接到一个门元件或模块的输出端；或用assign语句对其赋值

❖ 常用的nets型变量

➢ wire，tri：连线类型（两者功能一致），可综合

➢ wor，trior：具有线或特性的连线（两者功能一致）

➢ wand，triand：具有线与特性的连线（两者功能一致）

➢ tri1，tri0：上拉电阻和下拉电阻

➢ supply1，supply0：电源（逻辑1）和地（逻辑0），可综合

wire型变量

❖wire型变量

➢ 最常用的nets型变量，常用来表示以assign语句赋值的组合逻辑信号。

➢ 模块中的输入/输出信号类型缺省为wire型——当对输入/输出信号不加以信号类型声明时，则输入/输出信号为wire型。

➢ 可用做任何方程式的输入，或“assign”语句和实例元件的输出。

格式

```
wire 变量名1,变量名2, …,变量名n;
```

【例】 将输入a赋值给wire型变量b

```
input a;
wire b;        /* 中间节点。若为output信号，则默认为wire型变量，不必单独声明 */
assign b=a;  //当a变化时，b立即随之变化
```

wire型向量（总线）

位宽为1位的变量称为标量，
位宽超过1位的变量称为向量。向量的宽度定义：
[MSB : LSB] /* MSB(Most Significant Bit，最高有效位），
LSB (Least Significant Bit，最低有效位） */

格式

wire[n-1:0] 变量名1,变量名2, …,变量名m;
或 wire[n:1] 变量名1,变量名2, …,变量名m;

每条总线位宽为n

共有m条总线

【例】 wire型向量

```
wire[7:0]  in,out;
assign out=in;  //将等号右边的值赋给等号左边的变量。
```


register型变量

2. register型变量

- 寄存器型变量（register型变量）对应具有状态保持作用的电路元件（如触发器、寄存器等）,常用来表示过程块语句（如initial，always，task，function）内的指定信号。
- 常用的register型变量
  - reg：常代表触发器、寄存器，可综合
  - integer：32位带符号整数型变量，可综合
  - real：64位带符号实数型变量，表示实数寄存器，用于仿真
  - time：无符号时间变量，用于对仿真时间的存储与处理

纯数学的抽象描述

register型变量与nets型变量的区别

- register型变量需要被明确地赋值，并且在被重新赋值前一直保持原值。
- register型变量必须通过过程赋值语句赋值！不能通过assign语句赋值！
- nets型变量必须通过assign语句赋值！不能通过过程赋值语句赋值！
- 在always、initial、task、function等过程块内被赋值的每个信号必须定义成register型！

reg型变量

- reg型变量
  - reg型变量是数字系统中存储设备的抽象，常用于具体的硬件描述，是最常用的寄存器型变量。
  - 它是在过程块中被赋值的信号，往往代表触发器（沿触发时），但不一定就是触发器（也可以是组合逻辑信号，电平触发时）！

格式

reg 变量名1,变量名2, ……,变量名n;

- reg型向量（总线）

```
reg[n-1:0] 变量名1,变量名2, …,变量名m;
或 reg[n:1] 变量名1,变量名2, …,变量名m;
```

【例】 reg[4:1] regc,regd; //regc,regd为4位宽的reg型向量
reg[0:7] data；//8位寄存器型变量，最高有效位是0，最低有效位是7

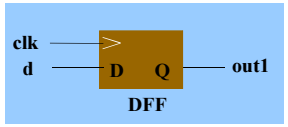
- 向量定义后可以采用多种使用形式（即赋值）

data=8'b00000000; data[5:3]=3'B111; data[7]=1；

reg型变量生成触发器和组合逻辑举例

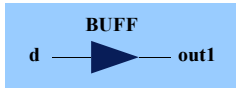
【 例 】在时钟沿触发的always块中，用reg型变量生成触发器

```
module  rw1( clk, d, out1) ;
        input clk, d ;
        output out1 ;
        reg out1 ;
        always @(posedge clk) //沿触发
          out1 <=  d;
endmodule
```



【例】使用电平触发，用reg型变量生成组合逻辑

```
module  rw2( clk, d, out1);
    input clk, d;
    output out1;
    reg out1;
    always @(d)  //电平触发
       out1 <= d;
endmodule
```



返回

Verilog HDL常用语句

一、结构声明语句
二、赋值语句
三、条件语句
四、循环语句
五、语句的顺序执行与并行执行

- 语句是构成Verilog HDL程序不可缺少的部分。
- Verilog HDL的语句包括赋值语句、条件语句、循环语句、结构声明语句和编译预处理语句等类型，每一类语句又包括几种不同的语句。
- 有些语句属于顺序执行语句，有些语句属于并行执行语句。

Verilog HDL常用语句

赋值语句	门基元赋值语句	
	连续赋值语句	
	过程赋值语句	
块语句	begin_end语句	
	fork_join语句	Quartus II不支持
条件语句	if_else语句	
	case语句	
循环语句	forever语句	
	repeat语句	
	while语句	
	for语句	
结构声明语句	initial语句	Quartus II不支持
	always语句	
	task语句	
	function语句	
编译预处理语句	‘define语句	
	‘include语句	Quartus II不支持
	‘timescale语句	Quartus II不支持

一、结构声明语句

- 具有某种独立功能的电路，均在过程块中描述，Verilog HDL的任何过程模块都是放在结构声明语句中，结构声明语句分为4种：
  - initial语句——沿时间轴只执行一次
  - always语句——不断重复执行，直到仿真结束
  - task语句——可在程序模块中的一处或多处调用
  - function语句——可在程序模块中的一处或多处调用

1、always块语句

- 包含一个或一个以上的声明语句(如:过程赋值语句、任务调用、条件语句和循环语句等），在仿真运行的全过程中，在定时控制下被反复执行。

规则

- always块通常带触发条件；
- 在always块中被赋值的只能是register型变量。
- 每个在仿真一开始便开始执行，当执行完块中最后一个语句，继续从always块的开头执行。

always块语句

格式

```
always @(敏感信号表达式)
  begin
    // 过程赋值语句;
    // if语句，case语句;
    // for语句，while语句，repeat语句;
    // tast语句、function语句;
  end
```

一个变量不能在多个always块中被赋值！

错误的写法：

```
always @ (posedge clk)
  q=q+1;
always @ (negedge reset)
  q=0;
```

正确的写法：

```
always @ (posedge clk or
negedge reset)
  begin
    if(!reset) q=0; //异步清零
    else q=q+1;
  end
```


2、initial语句

- initial语句是面向模拟仿真的过程语句，通常不能被逻辑综合工具支持。initial块内的语句仅执行一次。

【例2.26】对各变量进行初始化。

格式

```
initial
  begin
    语句1;
    语句2;
    ……
    语句n;
  end
```

```
parameter size=16;
reg[3:0] addr;
reg reg1;
reg[7:0] memory[0:15];
initial
  begin
    reg1 = 0;
    for(addr=0;addr<size;addr=addr+1);
      memory[addr]=0;
  end
```

用途

- 在仿真的初始状态对各变量进行初始化；
- 在测试文件中生成激励波形（如时钟信号）作为电路的仿真信号。

3、task语句

- task语句用来由用户定义任务，任务类似高级语言中的子程序，用来单独完成某项具体任务，并可以被模块或其他任务调用。
- 当希望能够对多个信号进行一些运算并输出多个结果（即有多个输出变量）时，宜采用任务结构。
- 常常利用任务来帮助实现结构化的模块设计，将批量的操作以任务的形式独立出来，使设计简单明了，而且便于调试。

任务定义

注意无端口列表！

```
task <任务名>;
  端口声明语句;
  数据类型声明语句;
  实现逻辑功能的语句;
endtask
```

任务调用

```
<任务名> (端口1,端口2,……);
```

端口名列表与任务定义中的I/O变量一一对应！

task语句使用注意事项

注1：任务的定义与调用必须在一个module模块内！
注2：任务被调用时，需列出端口名列表，端口名的排序与类型必须与任务定义中的I/O变量相一致！
注3：一个任务可以调用其他任务和函数。

【例】任务的定义与调用。

任务定义

```
task my_task;
  input a,b;
  inout c;
  output d,e;
    ……
    <语句>      //执行任务工作相应的语句
    ……
  c = foo1;
  d = foo2;     //对任务的输出变量赋值
  e = foo3;
endtask
```

任务调用

```
my_task (v,w,x,y,z);
```

- 当任务启动时，由v、w和x传入的变量赋给了a、b和c；
- 当任务完成后，输出通过c、d和e赋给了x、y和z。

4、function语句

- function语句用来定义函数，函数的目的是通过返回一个用于某表达式的值，来响应输入信号。适于对不同变量采取同一运算的操作。
- 函数在模块内部定义，通常在本模块中调用，也能根据按模块层次分级命名的函数名从其他模块调用。而任务只能在同一模块内定义与调用！

函数定义

只能是input

```
function <返回值位宽或类型说明> 函数名;
  端口声明;
  局部变量定义;
  其他语句;
endfunction
```

缺省则返回1位reg型数据

函数调用

```
<函数名> (<表达式1>, <表达式2>, ..... )
```

与函数定义中的输入变量对应！

function语句举例

函数在综合时被理解成具有独立运算功能的电路，每调用一次函数，相当于改变此电路的输入，以得到相应的计算结果。

```
【例2.29】 利用函数对一个8位二进制数中为0的位进行计数。
function[3:0] get0; //函数的定义，计算x中0的个数
  input [7:0] x;
  reg[3:0] count; integer i;
  begin count=0;
    for(i=0;i<=7;i=i+1) //循环核对x中的每一位
      if(x[i]==1'b0) count=count+1;
    get0 = count;  //将运算结果赋给与函数同名的内部寄存器
  endfunction
assign number = get0(rega); //对函数的调用
```

只有输入变量

内部寄存器

对应函数的输入变量

函数的调用是通过将函数作为调用函数的表达式中的操作数来实现的！

二、赋值语句

❖赋值语句分为3类：

1、门基元赋值语句（门元件例化）

```
基本逻辑门关键字 (门输出, 门输入1, 门输入2, ..., 门输入n);
```

- 基本逻辑门关键字是Verilog HDL预定义的逻辑门，包括and、or、not、xor、nand、nor等；圆括弧中内容是被描述门的输出和输入信号。
- 例如，具有a、b、c、d这4个输入和y为输出的与非门的门基元赋值语句为nand (y,a,b,c,d);
  该语句与assign y = ! (a && b && c && d);等效

2、连续赋值语句（assign语句）

用于对wire型变量赋值，是描述组合逻辑最常用的方法之一。

```
assign 赋值变量 = 表达式;
```

- 【例】 4输入与非门
  assign y = ! (a && b && c && d);
- 连续赋值语句的“=”号两边的变量都应该是wire型变量。
- 在执行中，输出y的变化跟随输入a、b、c、d的变化而变化，反映了信息传送的连续性。

```
【例】 2选1多路选择器
module mux2_1(out,a,b,sel);
    input a,b,sel; output out;    //输入、输出信号默认为wire型变量
    assign out =( sel==0) ? a:b; //若sel为0，则out=a；否则out=b
endmodule
```

3、过程赋值语句

3、过程赋值语句

用于对reg型变量赋值，过程赋值语句出现在initial和always块语句中，有两种赋值方式：

- 阻塞（blocking）赋值方式：

  赋值符号为=，如 b = a ;

  赋值变量 = 表达式;

- 非阻塞（non-blocking）赋值方式：赋值符号为<=，如 b <= a ;

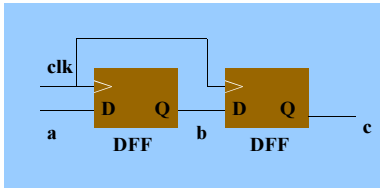
  赋值变量 <= 表达式;

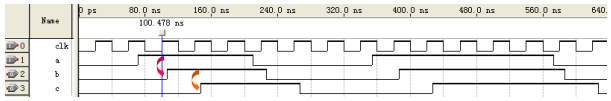
非阻塞赋值与阻塞赋值的区别

（1）非阻塞赋值方式

```
always @(posedge clk)
    begin
      b <= a;
      c <= b;
    end
```

非阻塞赋值在块结束时才完成赋值操作！





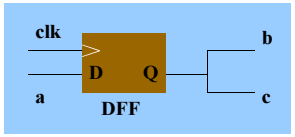
- c的值比b的值落后一个时钟周期！
- 若块内有多条赋值语句，则在块结束时同时赋值。
- 多条非阻塞赋值语句并行执行！

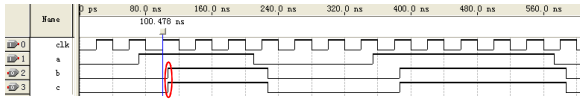
非阻塞赋值与阻塞赋值的区别（续）

（2）阻塞赋值方式

```
always @(posedge clk)
    begin
      b = a;
      c = b;
    end
```

阻塞赋值在该语句结束时就完成赋值操作！





- c的值与b的值一样！
- 在一个块语句中，如果有多条阻塞赋值语句，在前面的赋值语句没有完成之前，后面的语句就不能被执行，就像被阻塞了一样，因此称为阻塞赋值方式。
- 多条阻塞赋值语句顺序执行！

三、条件语句

条件语句分为两种：if-else语句和case语句；
它们都是顺序语句，应放在“always”块内！

对于每个判定只有两个分支

1、 if-else语句

- 判定所给条件是否满足，根据判定的结果（真或假）决定执行给出的两种操作之一。
- if-else语句有3种形式，格式与C语言中的if-else语句类似
  - 其中“表达式”为逻辑表达式或关系表达式，或一位的变量。
  - 若表达式的值为0、或x、或z，则判定的结果为“假”；若为1，则结果为“真”。
  - 执行的语句可为单句，也可为多句；多句时一定要用“begin_end”语句括起来，形成一个复合块语句。

if-else语句的表示方式

方式1（非完整性IF语句）：

```
if (表达式) 语句1;
```

方式2（二重选择的IF语句）：

```
if (表达式1) 语句1;
else         语句2;
```

方式3（多重选择的IF语句）：

```
if (表达式1) 语句1;
else if (表达式2) 语句2;
else if (表达式3) 语句3;
   …
else  语句n;
```

适于对不同的条件，执行不同的语句

❖ else子句不能作为语句单独使用，它是if语句的一部分，必须与if配对使用!

❖允许一定形式的表达式简写方式，如：

- if(expression) 等同于if(expression = = 1)
- if(! expression) 等同于if(expression ! = 1)

2、case语句

多分支选择语句

适于对同一组控制信号取不同的值时，输出取不同的值！

❖ 当敏感表达式取不同的值时，执行不同的语句。

❖ 功能：当某个或某组（控制）信号取不同的值时，给另一个（输出）信号赋不同的值。常用于多条件译码电路（如译码器、数据选择器、状态机、微处理器的指令译码）！

case语句的语法格式

```
case（敏感表达式）
   值1：语句1;
   值2：语句2;
    …
   值n：语句n;
   default: 语句n+1;
endcase
```

❖说明：

- 其中“敏感表达式”又称为“控制表达式”，通常表示为控制信号的某些位。当有多个信号时，可用位拼接符将它们连接起来：
- 【例】 case({D3,D2,D1,D0})
- 值1~值n称为分支表达式，用控制信号的具体状态值表示，因此又称为常量表达式。
- default项可有可无，一个case语句里只能有一个default项!
- 值1~值n必须互不相同，否则矛盾。
- 值1~值n的位宽必须相等，且与控制表达式的位宽相同。

四、循环语句

❖循环语句用来控制语句的执行次数。分为4种：

- for语句——有条件的循环语句。通过3个步骤来决定语句的循环执行：
  - （1）给控制循环次数的变量赋初值。
  - （2）判定循环执行条件，若为假则跳出循环；若为真，则执行指定的语句后，转到第（3）步。
  - （3）修改循环变量的值，返回第（2）步。
- repeat语句——连续执行一条语句n次
- while语句——执行一条语句直到某个条件不满足。首先判断循环执行条件表达式是否为真，若为真，则执行后面的语句或语句块，直到条件表达式不为真；若不为真，则其后的语句一次也不被执行！
- forever语句——无限连续地执行语句，可用disable语句中断！多用在initial块中，以生成时钟等周期性波形

1、for语句

功能：一般用途的循环语句，允许一条或更多的语句被重复地执行。

一般形式

```
for (表达式1;表达式2;表达式3) 语句
```

简单应用形式

```
for (循环指针 = 初值; 循环指针 < 终值; 循环指针 = 循环指针 + 步长值)
   begin 执行语句; end
```

两条语句

for语句比while语句简洁！

规则：当for循环开始执行时，循环变量已赋予初值。在每一次循环执行之前（包括第一次），都必须检查表达式2（循环执行条件），若它为假（0、x或z），则立刻退出循环。而在每一次循环执行之后，都要使循环变量增值。

for语句举例

【例2.34】用for语句描述的7人投票表决器：若超过4人（含4人）投赞成票，则表决通过。

```
module  vote7 ( pass,vote );
    output pass;
    input [6:0] vote;
    reg[2:0] sum;      //sum为reg型变量，用于统计赞成的人数
    integer i;              // 循环变量
    reg pass;
    always @(vote)
      begin
        sum = 3'b000;                    //sum初值为0
        for(i = 0;i<=6;i = i+1)          //for语句
          if(vote[i])     sum = sum+1; //只要有人投赞成票，则 sum加1
        if(sum >=3'd4)    pass =1'b 1;  //若超过4人赞成，则表决通过
        else              pass =1'b 0;
      end
endmodule
```

将循环变量定义为整型

2、repeat语句

只有部分综合工具可以综合此语句！

❖**功能**：把一条或多条语句连续执行指定的次数。

❖**规则**：重复执行的次数由循环次数表达式的值决定，若该值为0、x或z，则不会重复执行。

格式

```
repeat (循环次数表达式) 语句
```

或

```
repeat (循环次数表达式)
  begin
    ……
  end
```

执行语句为多条语句

3、 while语句

❖ **功能**：有条件地执行一条或多条语句。只要循环执行条件表达式为真，则循环语句就重复执行！

❖ **规则**：首先判断循环执行条件表达式是否为真。若为真，则执行后面的语句或语句块；然后再回头判断循环执行条件表达式是否为真，若为真，再执行一次后面的语句；如此不断，直到条件表达式不为真。

格式

```
while (循环执行条件表达式)
  begin
    ……
  end
```

while语句通常用在测试文件中！

1.首先判断循环执行条件表达式是否为真，若不为真，则其后的语句一次也不被执行！

2.在执行语句中，必须有一条改变循环执行条件表达式的值的语句！

3.while语句只有当循环块有事件控制（即@（posedge clock））时才可综合！

while语句举例

【例2.37】用while语句对一个8位二进制数中值为1的位进行计数

```
module  count1s_while ( count,rega,clk );
    output[3:0] count;
    input [7:0]  rega;
    input clk;
    reg[3:0]     count;
    always @(posedge clk)
      begin:count1
        reg[7:0] tempreg;          //用作循环执行条件表达式
        count = 0;                 // count初值为0
        tempreg = rega;            // tempreg 初值为rega
        while(tempreg)             // 若tempreg非0，则执行以下语句
          begin
            if(tempreg[0])     count = count+1;
                                   //只要tempreg最低位为1，则 count加1
            tempreg = tempreg >>1; //右移1位
          end
      end
endmodule
```

如何用for语句改写此程序呢？

改变循环执行条件表达式的值

4、 forever语句

❖功能：无条件连续执行forever后面的语句或语句块。

格式

```
forever
  begin
    ……
  end
```

forever语句通常用在测试文件中！

- forever循环应包括定时控制或能够使其自身停止循环，否则循环将无限进行下去！
- 常用在测试文件中产生周期性的波形，作为仿真激励信号。
- 可综合性：尽管Quartus II支持该语句，但一般情况下是不可综合的！如果forever循环被@(posedge clock)形式的时间控制打断，则是可综合的。

五、语句的顺序执行与并行执行

❖Verilog HDL中有顺序执行语句和并行执行语句之分。

1、语句的顺序执行

- 在 “always”模块内，对于阻塞赋值语句，逻辑按书写的顺序执行，若随意颠倒赋值语句的书写顺序，可能导致不同的结果！（见下页例子）。
- 而对于非阻塞赋值语句，是并发执行的。
- 注意阻塞赋值语句当本语句结束时即完成赋值操作！

语句的顺序执行举例

【例2.40】顺序执行模块1。

```
module serial1(q,a,clk);
    output q,a;
    input clk;
    reg q,a;
    always @(p
      begin
        q=~q; //阻塞赋值语句
        a=~q;
      end
endmodule
```

对前一时刻的q值取反

对当前时刻的q值反

a和q的波形反相！

【例2.41】顺序执行模块2。

```
module serial2(q,a,clk);
    output q,a;
    input clk;
    reg q,a;
    always @
      begin
        a=~q;
        q=~q;
      end
endmodule
```

对前一时刻的q值取反

对前一时刻的q值取反

a和q的波形完全相同！

2、语句的并行执行

- 多个“always”模块、“assign”语句、实例元件调用、“always”模块内的非阻塞赋值语句都是并行执行的！
- 它们在程序中的先后顺序对结果并没有影响。
- 【例2.42】将两条赋值语句分别放在两个“always”模块中，若颠倒两个“always”模块顺序，对仿真结果没有影响，同【例2.41】——q和a的波形完全一样。

【例2.42】并行执行模块。

```
module parall1(q,a,clk);
   output q,a;
   input clk;
   reg q,a;
   always @(posedge clk)
     begin
       q=~q;
     end
   always @(posedge clk)
     begin
       a=~q;
     end
endmodule
```


不同抽象级别的Verilog HDL模型

- 用Verilog HDL描述的电路称为该设计电路的Verilog HDL模型。
- 一个复杂电路的完整Verilog HDL模型由若干个Verilog HDL模块构成，每个模块由若干的子模块构成——可分别用不同抽象级别的Verilog HDL描述。
- 在同一个Verilog HDL模块中可有多种级别的描述。
  - 系统级(system level): 用高级语言结构（如case语句）实现的设计模块外部性能的模型；
  - 算法级(algorithmic level): 用高级语言结构实现的设计算法模型（写出逻辑表达式）；
  - RTL级(register transfer level): 描述数据在寄存器之间流动和如何处理这些数据的模型；
  - 门级(gate level): 描述逻辑门（如与门、非门、或门、与非门、三态门等）以及逻辑门之间连接的模型；
  - 开关级(switch level): 描述器件中三极管和储存节点及其之间连接的模型。

不同抽象级别的Verilog HDL模型

Verilog HDL的行为描述和结构描述

- Verilog HDL具有行为描述和结构描述功能
  - 行为描述是对设计电路的逻辑功能的描述，并不用关心设计电路使用哪些元件以及这些元件之间的连接关系。行为描述通过描述电路的行为特性来设计电路，属于高层次的描述方法，包括系统级、算法级和寄存器传输级等3种抽象级别。
  - 结构描述是对设计电路的结构进行描述，即描述设计电路使用的元件及这些元件之间的连接关系。结构描述通过调用电路元件（如逻辑门，甚至晶体管）来构建电路，属于低层次的描述方法，包括门级和开关级2种抽象级别。
- 应重点掌握高层次描述方法，但门级描述在一些电路设计中也有一定的实际意义（系统速度快）。

不同抽象级别的Verilog HDL模型

- 采用的抽象级别越高，设计越容易，程序代码越简单；但耗用器件资源更多。对特定综合器，可能无法将某些抽象级别高的描述转化为电路！
- 基于门级描述的硬件模型不仅可以仿真，而且可综合，且系统速度快。
- 一般用系统级、算法级（写出逻辑表达式）或RTL级来描述逻辑功能，尽量避免用门级描述，除非对系统速度要求比较高的场合才采用门级描述。

第二部分：组合逻辑

一. 逻辑门电路
二. 布尔代数及其门电路实现
三. Verilog HDL介绍
1. Verilog HDL基本结构
2. Verilog HDL的词法
3. Verilog HDL常用语句
4. 不同抽象级别的Verilog HDL模型
四. 基本组合逻辑部件设计
1. 运算单元电路
2. 编码器/译码器
3. 多路选择器

组合逻辑电路的结构和特点

❖按照逻辑功能的不同特点，数字电路分为两大类：
- 组合逻辑电路和时序逻辑电路

❖组合逻辑电路是将逻辑门以一定的方式组合在一起，使其具有一定逻辑功能的数字电路。

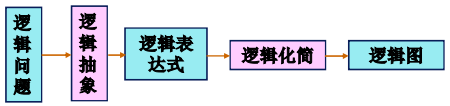
❖它是一种无记忆电路——任一时刻的输出信号仅取决于该时刻的输入信号，而与信号作用前电路原来所处的状态无关。

❖组合逻辑电路的特点
- 由逻辑门电路组成
- 输出不能再直接反馈到输入（不能有环路）和存储电路
- 当时的输出仅由当时的输入决定——速度快

组合逻辑电路的设计方法

❖ 组合逻辑电路的设计——根据给定的功能要求，采用某种设计方法，得到满足功能要求且最简单的组合逻辑电路。

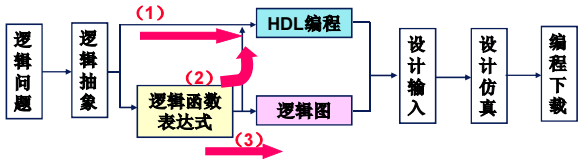
❖组合逻辑电路的手工设计方法
- 逻辑抽象——确定输入、输出变量，分析因果关系，列出真值表
- 写出逻辑函数表达式——根据真值表写出逻辑函数的标准表达式
- 逻辑化简——用公式化简法或卡诺图化简法化简为最简逻辑函数表达式
- 绘逻辑图——根据最简逻辑函数表达式画出原理图



组合逻辑电路的自动设计方法

❖ 基于HDL和EDA工具的组合逻辑电路的设计方法
- 逻辑抽象——确定输入、输出变量，列出真值表（复杂系统也可不写出真值表，而直接用HDL的系统级描述方式）
- 写出逻辑表达式——根据真值表写出逻辑函数的标准表达式
- HDL编程——如用case语句、if-else语句，assign语句

❖ 有3种途径
- 逻辑抽象→HDL编程（系统级描述，如用case语句或if-else语句）
- 逻辑抽象→写出逻辑函数表达式→HDL编程（算法级描述，assign语句）
- 逻辑抽象→写出逻辑函数表达式→绘逻辑图（适于简单电路）

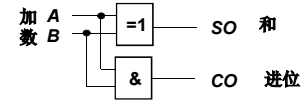


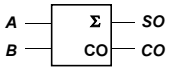
算术运算电路

- 常用的组合逻辑电路有算术运算电路、 编码器、译码器、数据选择器、数值比较器、奇偶校验器等
- 算术运算电路是能完成二进制数算术运算的器件
- 半加器和全加器是算术运算电路的基本单元电路

1、半加器

- 半加器——能对两个1位二进制数进行相加求和，并向高位进位的逻辑电路。
- 特点：不考虑来自低位的进位。





$SO = A \oplus B$

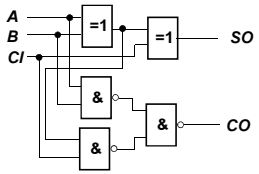
$CO = AB$

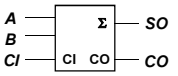
真值表

A	B	SO	CO
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

全加器

1位全加器——能对两个1位二进制数进行相加并考虑低位来的进位、求得和并向高位进位的逻辑电路称为全加器。
特点：考虑来自低位的进位的加法运算电路





$$SO = A \oplus B \oplus CI$$
$$= (A \oplus B)\cdot \overline{CI} + \overline{A \oplus B}\cdot CI$$
$$= (A\overline{B} + \overline{A}B)\overline{CI} + (\overline{A\overline{B} + \overline{A}B})CI$$
$$= A\overline{B}\,\overline{CI} + \overline{A}B\,\overline{CI} + \overline{A\overline{B}}\,CI + AB\,CI$$

$$CO = \overline{(A \oplus B)CI \cdot \overline{AB}}$$
$$= (A\overline{B} + \overline{A}B)CI + AB$$
$$= A\overline{B}CI + \overline{A}BCI + AB$$

真值表

A B CI	SO	CO
0 0 0	0	0
0 0 1	1	0
0 1 0	1	0
0 1 1	0	1
1 0 0	1	0
1 0 1	0	1
1 1 0	0	1
1 1 1	1	1

1位全加器的HDL设计

方法一：根据全加器的功能列出1位全加器的真值表，由真值表推出输出的逻辑表达式，然后用assign语句建模（算法级描述）

$$SO = \overline{A}\,\overline{B}CI + \overline{A}B\,\overline{CI} + A\,\overline{B}\,\overline{CI} + ABCI$$

$$CO = \overline{A}BCI + A\overline{B}CI + AB\overline{CI} + ABCI$$

真值表

A B CI	SO	CO
0 0 0	0	0
0 0 1	1	0
0 1 0	1	0
0 1 1	0	1
1 0 0	1	0
1 0 1	0	1
1 1 0	0	1
1 1 1	1	1

```
1位全加器Verilog HDL源程序（assign建模）
module adder_1(A,B,CI,SO,CO);
    input    A,B,CI;
    output   SO,CO;
    assign   SO = (!A&&!B&&CI)||(!A&&B&&!CI)||
                  (A&&!B&&!CI)||(A&&B&&CI);
    assign   CO =(!A&&B&&CI)||(A&&!B&&CI)||
                  (A&&B&&!CI)||(A&&B&&CI);
endmodule
```


1位全加器Verilog HDL源程序（系统级抽象）

方法二：采用行为描述方式的系统级抽象根据逻辑功能定义直接描述，程序更简洁！

```
module adder_2(A,B,CI,SO,CO);
    input     A,B,CI;
    output    SO,CO;
    assign    {CO,SO} = A+B+CI;
endmodule
```

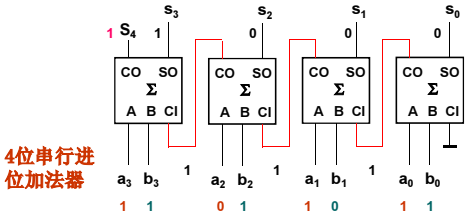
- 这里用位拼接运算符 “{ }”将进位与算术和拼接在一起成为一个2位数

多位加法器

- 能实现多位二进制数相加的电路称为多位加法器（Adder）。
- 由多个1位全加器可以扩展成多位加法器。
- 按进位方式不同，多位加法器分为串行进位加法器和并行进位加法器。

（1）串行进位加法器

- 串行进位加法器是依次将低位全加器的进位输出端*CO*接到高位全加器的进位输入端*CI*，加法从低位开始，高位全加必须等低位进位来到后才能进行，因此完成加法的时间较长。
- 优点：电路比较简单，连接方便；缺点：运算速度不高。



4位串行进位加法器

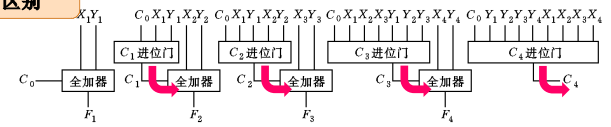
$A = a_3a_2a_1a_0$(=1011)
$B = b_3b_2b_1b_0$(=1101)
$S = A + B = (s_4)\ s_3s_2s_1s_0$
=1 1000

并行进位加法器（超前进位加法器）

（2）并行进位加法器

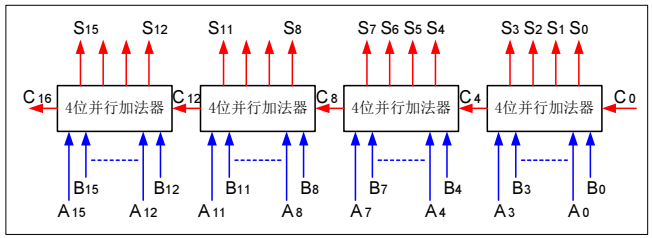
- 串行进位加法器运算速度不高，且延迟级数与位数成正比。
- 考虑设置专用的进位形成电路同时产生各位的进位Cn，进位输入由专门的“进位门”综合所有低位的加数、被加数及最低位进位来提供，这样构成的加法器称为并行进位加法器，又称超前进位或快速进位加法器。
- 各位进位由所有低位的加数、被加数及最低位进位C_0来决定，而与前一级加法器的进位输出无关，多位加法器的加法运算可以同时进行，因此完成加法的时间较快——提高了运算速度。

与串行进位加法器的本质区别



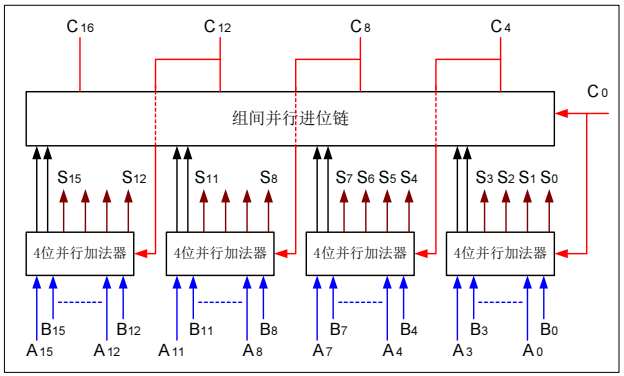
并行进位

- 分组并行进位加法器（组内并行，组间传递）

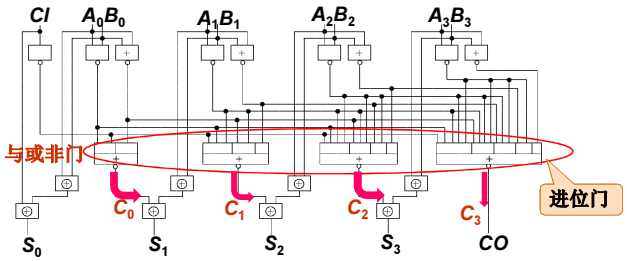


并行进位

- 分组并行进位加法器（组内并行，组间并行）



4位超前进位加法器74283



$$S_0 = A_0 \oplus B_0 \oplus CI$$

$$C_0 = \overline{\overline{A_0 + B_0} + \overline{A_0 \cdot B_0} \cdot \overline{CI}}$$

多位加法器的设计

- 用Verilog HDL行为描述方式很容易编写出任意位数的加法器电路。
- 8位加法器的Verilog HDL源程序adder_8.v:

```
module adder_8(a,b,cin,sum,cout);
    parameter   width=8;
    input [width-1:0]     a,b;
    input                 cin;
    output [width-1:0]    sum;
    output                cout;
    assign          {cout,sum} = a+b+cin;
endmodule
```

- 这里用parameter常量width表示加法器的位数，通过修改width，可以方便地实现不同位宽的加法器。

加减法运算

- 原则（以定点整数为例说明）

$$[A+B]_{补} = [A]_{补} + [B]_{补}$$

$$[A-B]_{补} = [A]_{补} + [-B]_{补}$$

- [X]补与[−X]补

若$[x]_{补} = x_0 x_1 x_2 \ldots x_{n-1}$

则$[-x]_{补} = \overline{x}_0 \overline{x}_1 \overline{x}_2 \ldots \overline{x}_{n-1} + 1$

所以有

$$[A-B]_{补} = [A]_{补} + \overline{[B]_{补}} + 1$$

加减法可共用一套加法器电路

阵列乘法器

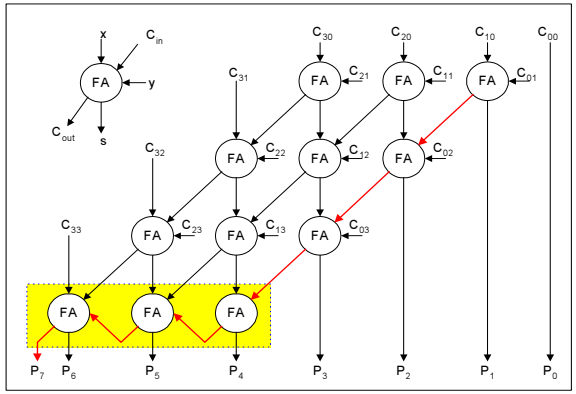
- 基本思路
  - 利用若干全加器，完全由硬件直接计算乘法结果
  - 以 4 位无符号数为例

$$\begin{array}{cccccccc} & & & & A_3 & A_2 & A_1 & A_0 \\ \times & & & & B_3 & B_2 & B_1 & B_0 \\ \hline & & & & C_{30} & C_{20} & C_{10} & C_{00} \\ & & & C_{31} & C_{21} & C_{11} & C_{01} & \\ & & C_{32} & C_{22} & C_{12} & C_{02} & & \\ + & C_{33} & C_{23} & C_{13} & C_{03} & & & \\ \hline P_7 & P_6 & P_5 & P_4 & P_3 & P_2 & P_1 & P_0 \end{array}$$

其中 $C_{ij} = A_i B_j$

阵列乘法器

❖实现电路



阵列乘法器

❖总结：

- 对于n位的阵列乘法，需全加器n(n-1)个
- 最长路径2(n-1)个全加器延时
- 最后的串性进位可采用先行进位加法器

数值比较器

❖数值比较器是一种关系运算电路，它可以对两个二进制数或二-十进制编码的数进行比较，得出大于、小于和相等的结果。

❖分为“等值”比较器和“量值”比较器，“等值”比较器只检验两个数是否相等；“量值”比较器不但检验两个数是否相等，而且还要检验两个数中哪个为大。

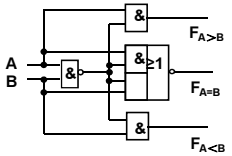
1、1位数值比较器

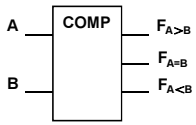
用来比较两个一位二进制数大小的电路。

$$F_{A>B} = A\overline{AB} = A(\overline{A}+\overline{B}) = A\overline{B}$$

$$F_{A<B} = B\overline{AB} = B(\overline{A}+\overline{B}) = \overline{A}B$$

$$F_{A=B} = \overline{A\overline{AB}+B\overline{AB}} = \overline{A\overline{B}+\overline{A}B} = AB+\overline{A}\overline{B}$$





真值表

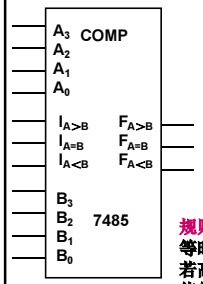
A B	$F_{A>B}$	$F_{A=B}$	$F_{A<B}$
0 0	0	1	0
0 1	0	0	1
1 0	1	0	0
1 1	0	1	0

4位数值比较器（7485）

2、4位数值比较器（7485）

用来比较两个4位二进制数大小的电路。

（1）逻辑符号



（2）功能表

级联输入端，用于芯片的扩展

A3 B3	A2 B2	A1 B1	A0 B0	$I_{A>B}$ $I_{A=B}$ $I_{A<B}$	$F_{A>B}$ $F_{A=B}$ $F_{A<B}$
A3>B3	X	X	X	X X X	1 0 0
A3<B3	X	X	X	X X X	0 0 1
A3=B3	A2>B2	X	X	X X X	1 0 0
A3=B3	A2<B2	X	X	X X X	0 0 1
A3=B3	A2=B2	A1>B1	X	X X X	1 0 0
A3=B3	A2=B2	A1<B1	X	X X X	0 0 1
A3=B3	A2=B2	A1=B1	A0>B0	X X X	1 0 0
A3=B3	A2=B2	A1=B1	A0<B0	X X X	0 0 1
A3=B3	A2=B2	A1=B1	A0=B0	a b c	a b c

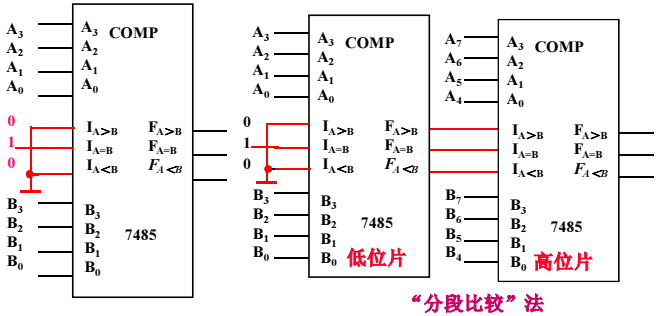
规则：从高位开始比较，高位不等时，数值的大小由高位决定；若高位相等，则再比较低位，数值的大小由低位比较结果决定。

若$A_3>B_3$ 则A>B；
若$A_3<B_3$ 则A<B；
若$A_3=B_3$ 则再比较低位

7485的使用与扩展方法

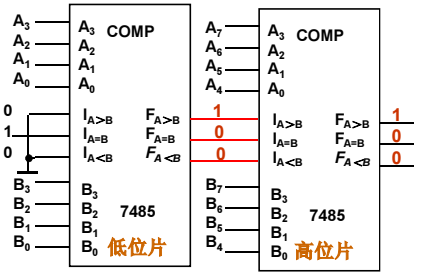
（3）使用与扩展方法

①单片使用——4位数值比较器　　②2片扩展——8位数值比较器



“分段比较”法

2片扩展——“分段比较”法

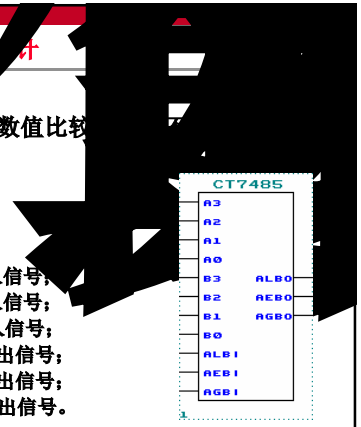


- 低位片和高位片并行工作，每片的比较仍是由高位到低位逐位进行
  - 若高4位数不相等，则由两个高4位数A7~A4与B7~B4的大小决定A和B的大小。

- 若高4位分别相等，则由两个低4位数A3~A0与B3~B0的大小决定A和B的大小：若A3~A0>B3~B0，则低位片的输出$F_{A>B}$、$F_{A=B}$、$F_{A<B}$为100，即高位片的级联输入$I_{A>B}$、$I_{A=B}$、$I_{A<B}$为100，由功能表的最后一行可以得出，高位片的输出$F_{A>B}$、$F_{A=B}$、$F_{A<B}$也为100，即A>B；同理，若A3~A0<B3~B0，则可推出A<B；若A3~A0=B3~B0，则可推出A=B。

数值比较器（7485）的HDL设计

- 可以方便地用HDL设计多位数值比较器，不必用扩展的方法
- 采用if-else语句
- 信号定义
  - A3～A0和B3～B0：两个4
  - ALBI（即$I_{A<B}$）：A小于B输入信号；
  - AEBI（即$I_{A=B}$）：A等于B输入信号；
  - AGBI（即$I_{A>B}$）：A大于B输入信号；
  - ALBO（即$F_{A<B}$）：A小于B输出信号；
  - AEBO（即$F_{A=B}$）：A等于B输出信号；
  - AGBO（即$F_{A>B}$）：A大于B输出信号。

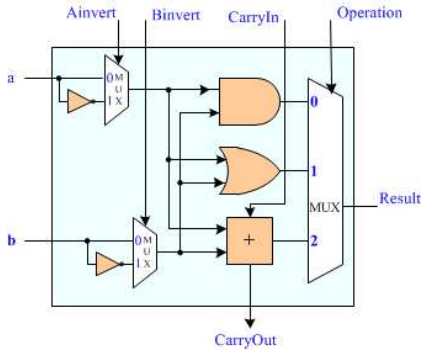


7485的Verilog HDL源程序

```
module CT7485(A3,A2,A1,A0,B3,B2,B1,B0,ALBI,AEBI,
        AGBI,ALBO,AEBO,AGBO);
   input        A3,A2,A1,A0,B3,B2,B1,B0,ALBI,AEBI,AGBI;
   output       ALBO,AEBO,AGBO;
   reg          ALBO,AEBO,AGBO;
   wire[3:0]    A_SIGNAL,B_SIGNAL;
   assign       A_SIGNAL = {A3,A2,A1,A0}; //拼接成4位wire型向量
   assign       B_SIGNAL = {B3,B2,B1,B0}; //拼接成4位wire型向量
   always
      begin
        if (A_SIGNAL > B_SIGNAL)
          begin  ALBO = 0; AEBO = 0; AGBO = 1;end
        else if (A_SIGNAL < B_SIGNAL)
          begin  ALBO = 1; AEBO = 0; AGBO = 0;end
        else // if(A_SIGNAL == B_SIGNAL)可省略
          begin  ALBO = ALBI; AEBO = AEBI; AGBO = AGBI;end
      end
endmodule
```

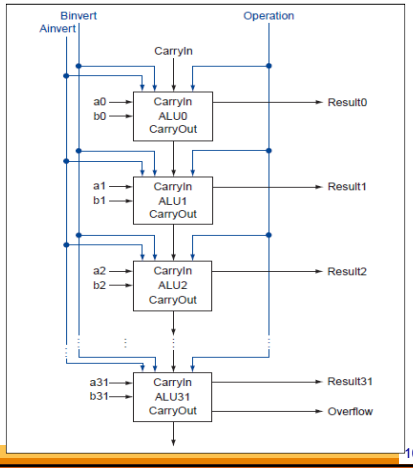
1位ALU

- 可执行1位与、或、或非、与非、加、减运算
  - AND运算
    - Ainvert=0
    - Binvert=0
    - Operation=0
  - NOR运算
    - Ainvert=1
    - Binvert=1
    - Operation=0
  - SUB运算
    - Ainvert=0
    - Binvert=1
    - CarryIn=1
    - Operation=2
  - . . .



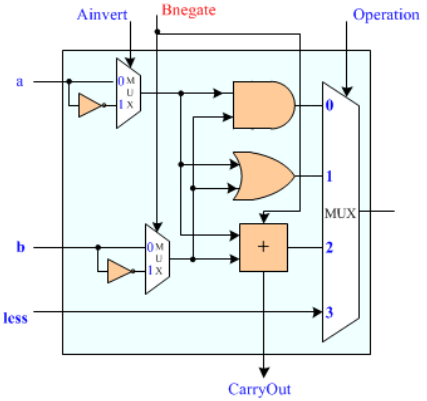
32位ALU

- 加法器采用串行仅为，也可以采用先行进位
- 减法的实现
  - Ainvert=0
  - Binvert=1
  - CarryIn=1
- Binvert和CarryIn可以合并成一个信号Bnegate



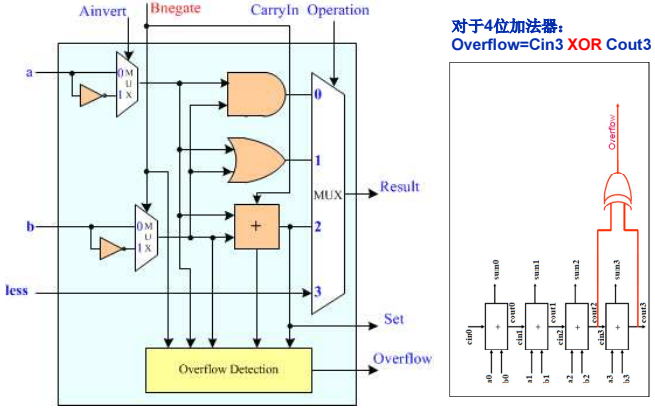
32位ALU

- 为比较功能（Less）增加一个输入



32位ALU

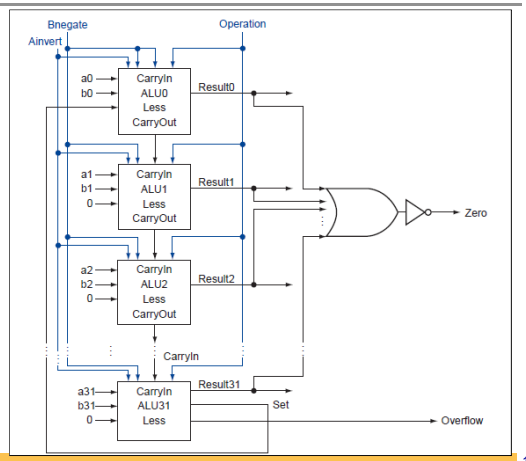
- 最高位（MSB）带溢出判断Overflow



对于4位加法器：
Overflow=Cin3 XOR Cout3

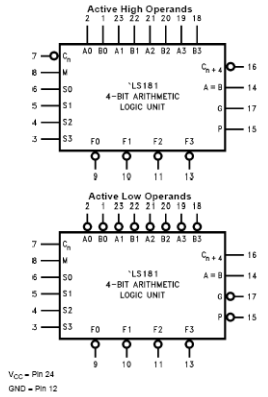
32位ALU

完整32位ALU

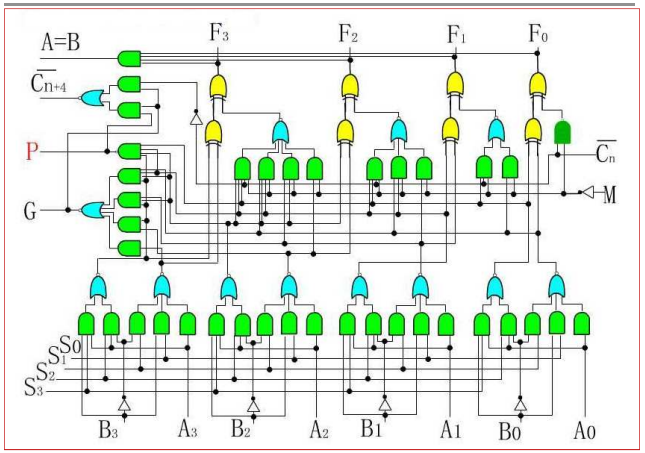


ALU芯片—DM74LS181N

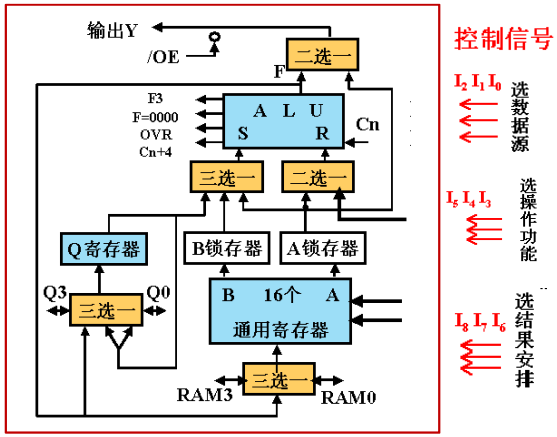
- 4位ALU
- 提供16种算术逻辑运算
- 两种工作模式：正逻辑和负逻辑



ALU芯片—DM74LS181N



AM2901(4位运算器器件)



AM2901（4位运算器器件）

编码			数据来源	
I2	I1	I0	R	S
L	L	L	A	Q
L	L	H	A	B
L	H	L	0	Q
L	H	H	0	B
H	L	L	0	A
H	L	H	D	A
H	H	H	D	Q
H	H	H	D	0

编码			运算
I5	I4	I3	功能
L	L	L	R + S
L	L	H	S − R
L	H	L	R − S
L	H	H	R ∨ S
H	L	L	$\overline{R} \wedge S$
H	L	H	$R \wedge \overline{S}$
H	H	L	$\overline{R \vee S}$
H	H	H	$\overline{R \wedge S}$

编 码			结 果 处 理		
I8	I7	I6	通用寄存器组	Q寄存器	Y输出
L	L	L		F→Q	F
L	L	H			F
L	H	L	F→B		A
L	H	H	F→B		F
H	L	L	F/2→B	Q/2→Q	F
H	L	H	F/2→B		F
H	H	L	2F→B	2Q→Q	F
H	H	H	2F→B		F

第二部分：组合逻辑

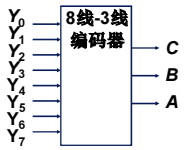
编码器

- 为了区分一系列不同的事物，将其中的每个事物用一组二值（0或1）代码表示；或者说，用二进制代码来表示特定信息——编码的含义。
- 将加在电路若干输入端中的某一个输入端的信号变换成相应的一组二进制代码输出的过程叫做编码。
- 实现编码功能的数字电路称为编码器（Encoder）。
- 编码器的作用是将某一时刻仅一个输入有效的多个输入的变量情况用较少的输出状态组合来表达，或者说将输入的每一个高、低电平信号编成一组对应的二进制代码，以便于后续的识别和处理。
- 通常有二进制编码器、BCD码编码器及优先编码器。

二进制编码器

- 用n位二进制代码对M=2^n个信号进行编码的电路叫二进制编码器。
  - 特点：任意一时刻只能对一个信号进行编码，即任何时刻只允许一个输入信号有效（低电平或高电平），而其余信号为无效电平，否则输出将发生混乱
  - n位二进制符号可以表示2^n种信息，称为2^n线-n线编码器。
  - 常用的有8线-3线编码器
- 如果在输入等于1时对输入信号进行编码，则称这类编码器为高电平输入有效，此时输出等于对应有效输入的编号的二进制编码；如果在输入等于0时对输入信号进行编码，则称这类编码器为低电平输入有效。

8线-3线编码器（高电平输入有效）



编码表

输入	C B A
Y_0	0 0 0
Y_1	0 0 1
Y_2	0 1 0
Y_3	0 1 1
Y_4	1 0 0
Y_5	1 0 1
Y_6	1 1 0
Y_7	1 1 1

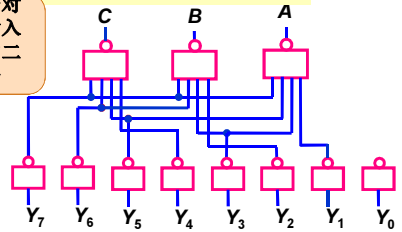
输出等于对应有效输入的编号的二进制编码

➢ 如果任何时刻输出编码仅对应一个有效输入信号，则对某位输出，在编码表中挑出所有为“1”的值，将其对应的输入信号相或，得到输出表达式。

$$C = Y_4 + Y_5 + Y_6 + Y_7 = \overline{\overline{Y_4} \cdot \overline{Y_5} \cdot \overline{Y_6} \cdot \overline{Y_7}}$$

$$B = Y_2 + Y_3 + Y_6 + Y_7 = \overline{\overline{Y_2} \cdot \overline{Y_3} \cdot \overline{Y_6} \cdot \overline{Y_7}}$$

$$A = Y_1 + Y_3 + Y_5 + Y_7 = \overline{\overline{Y_1} \cdot \overline{Y_3} \cdot \overline{Y_5} \cdot \overline{Y_7}}$$



8线-3线编码器的真值表

真值表（高电平输入有效）

Y_7 Y_6 Y_5 Y_4 Y_3 Y_2 Y_1 Y_0	C B A
0 0 0 0 0 0 0 1	0 0 0
0 0 0 0 0 0 1 0	0 0 1
0 0 0 0 0 1 0 0	0 1 0
0 0 0 0 1 0 0 0	0 1 1
0 0 0 1 0 0 0 0	1 0 0
0 0 1 0 0 0 0 0	1 0 1
0 1 0 0 0 0 0 0	1 1 0
1 0 0 0 0 0 0 0	1 1 1
0 0 0 0 0 0 1 1	x x x
0 0 0 0 0 1 1 1	x x x
约束项……	……
1 1 1 1 1 1 1 1	x x x

➢ 利用最小项推导法写出各输出的逻辑函数表达式

$$C = \overline{Y}_7\overline{Y}_6\overline{Y}_5Y_4\overline{Y}_3\overline{Y}_2\overline{Y}_1\overline{Y}_0 + \overline{Y}_7\overline{Y}_6Y_5\overline{Y}_4\overline{Y}_3\overline{Y}_2\overline{Y}_1\overline{Y}_0 + \overline{Y}_7Y_6\overline{Y}_5\overline{Y}_4\overline{Y}_3\overline{Y}_2\overline{Y}_1\overline{Y}_0 + Y_7\overline{Y}_6\overline{Y}_5\overline{Y}_4\overline{Y}_3\overline{Y}_2\overline{Y}_1\overline{Y}_0$$

➢ 如果任何时刻Y_7~Y_0中仅有一个输入取值为1，即输入变量取值的组合仅有表中的前8种状态，则输入变量为其他取值下输出等于1的那些最小项均为约束项。利用这些约束项化简上式，得到：

$$C = Y_4 + Y_5 + Y_6 + Y_7$$

8线-3线编码器的HDL设计

❖方法一：根据8线-3线编码器的功能列出真值表，由真值表推出输出的逻辑表达式，然后用assign语句建模（算法级描述）

```
module encoder8_3(Y0,Y1,Y2,Y3,Y4,Y5,Y6,Y7, C,B,A);
    input   Y0,Y1,Y2,Y3,Y4,Y5,Y6,Y7;
    output  C,B,A;
    assign   C =! (!Y4&&!Y5&&!Y6&&!Y7);
    assign   B =! (!Y2&&!Y3&&!Y6&&!Y7);
    assign   A =! (!Y1&&!Y3&&!Y5&&!Y7);
endmodule
```

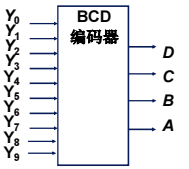
还可以写为？

❖方法二：根据逻辑功能定义，采用case语句直接描述，设计过程更简单！

```
reg      C,B,A;
always
  case ({Y0,Y1,Y2,Y3,Y4,Y5,Y6,Y7})
         'b10000000 : {C,B,A} = 0;
         'b01000000 : {C,B,A} = 1;
         ……
         'b00000001 : {C,B,A} = 7;
          default : {C,B,A} = 3'bx;
  endcase
```


BCD码编码器

➢ BCD码编码器就是用二进制码表示十进制数的编码器，也称为二-十进制编码器，或称为10线-4线编码器。
➢ 用4位二进制代码对十进制数的10个数码进行编码。
➢ BCD有多种编码方式：8421BCD、2421BCD或余3BCD。
➢ 通常用8421BCD来表示十进制数，构成8421BCD编码器。



高电平输入有效

➢ 10个输入端，分别接代表十进制数0~9的10个按键

编码表

输入	D C B A
Y_0	0 0 0 0
Y_1	0 0 0 1
Y_2	0 0 1 0
Y_3	0 0 1 1
Y_4	0 1 0 0
Y_5	0 1 0 1
Y_6	0 1 1 0
Y_7	0 1 1 1
Y_8	1 0 0 0
Y_9	1 0 0 1

$$D = Y_8 + Y_9 = \overline{\overline{Y_8} \cdot \overline{Y_9}}$$

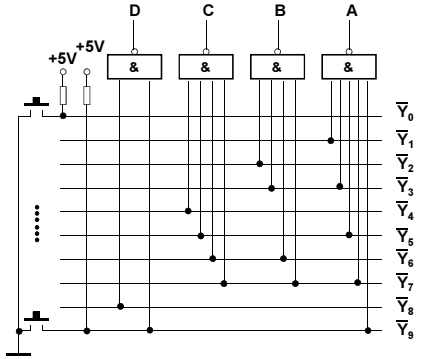
$$C = Y_4 + Y_5 + Y_6 + Y_7 = \overline{\overline{Y_4} \cdot \overline{Y_5} \cdot \overline{Y_6} \cdot \overline{Y_7}}$$

$$B = Y_2 + Y_3 + Y_6 + Y_7 = \overline{\overline{Y_2} \cdot \overline{Y_3} \cdot \overline{Y_6} \cdot \overline{Y_7}}$$

$$A = Y_1 + Y_3 + Y_5 + Y_7 + Y_9 = \overline{\overline{Y_1} \cdot \overline{Y_3} \cdot \overline{Y_5} \cdot \overline{Y_7} \cdot \overline{Y_9}}$$

8421BCD编码器的逻辑图

根据逻辑表达式可以直接画出逻辑图



$$D=\overline{\overline{Y}_8\cdot\overline{Y}_9}$$

$$C=\overline{\overline{Y}_4\cdot\overline{Y}_5\cdot\overline{Y}_6\cdot\overline{Y}_7}$$

$$B=\overline{\overline{Y}_2\cdot\overline{Y}_3\cdot\overline{Y}_6\cdot\overline{Y}_7}$$

$$A=\overline{\overline{Y}_1\cdot\overline{Y}_3\cdot\overline{Y}_5\cdot\overline{Y}_7\cdot\overline{Y}_9}$$

低电平输入有效

8421BCD编码器的Verilog HDL源程序

- 根据逻辑功能定义，直接采用case 语句描述——设计过程最简单！
- 假设高电平输入有效
- Y0=1时，DCBA=0000；
  Y1=1时，DCBA=0001；
  ……
  Y9=9时，DCBA=1001。

```
module bcd8421_3(Y0,Y1,Y2,Y3,Y4,Y5,Y6,Y7,Y8,Y9,D,C,B,A);
   input          Y0,Y1,Y2,Y3,Y4,Y5,Y6,Y7,Y8,Y9;
   output D,C,B,A;
   reg            D,C,B,A;
   always
     begin
       case ({Y9,Y8,Y7,Y6,Y5,Y4,Y3,Y2,Y1,Y0})
            10'b00_0000_0001 : {D,C,B,A} = 0;
            10'b00_0000_0010 : {D,C,B,A} = 1;
            10'b00_0000_0100 : {D,C,B,A} = 2;
            10'b00_0000_1000 : {D,C,B,A} = 3;
            10'b00_0001_0000 : {D,C,B,A} = 4;
            10'b00_0010_0000 : {D,C,B,A} = 5;
            10'b00_0100_0000 : {D,C,B,A} = 6;
            10'b00_1000_0000 : {D,C,B,A} = 7;
            10'b01_0000_0000 : {D,C,B,A} = 8;
            10'b10_0000_0000 : {D,C,B,A} = 9;
            default : {D,C,B,A} = 4'bxxxx;
       endcase
     end
endmodule
```

将输入最高位写在最左边；标明位宽；用下划线分隔多位数字

优先编码器

- 二进制编码器要求任何时刻只允许有一个输入信号有效，否则输出将发生混乱——当同时有多个输入信号有效时不能使用二进制编码器！
- 优先编码器可以避免这种情况发生。优先编码器事先对所有输入信号进行优先级别排序，允许两位以上的输入信号同时有效；但任何时刻只对优先级最高的输入信号编码，对优先级别低的输入信号则不响应，从而保证编码器可靠工作。
- 如有两个或两个以上的输入有效时，只对优先级最高的输入信号进行编码的编码器称为优先编码器。
  - 优点：当有两个或两个以上的输入有效时，输出不会发生混乱。
  - 广泛应用于计算机的优先中断系统、键盘编码系统中。
- 74LS148——8线-3线优先编码器，8个输入信号，低电平有效；3个输出端，反码输出
- 74LS147——10线-4线优先编码器，10个输入信号，低电平有效；4个输出端，反码输出

优先编码器 （74147）的设计

10线－4线优先编码 器CT74147的输 入信号为 $\overline{I}_0 \sim \overline{I}_9$，$\overline{I}_9$的优先权最高，$\overline{I}_0$最低。

4线输出信号为 $\overline{Y}_3 \sim \overline{Y}_0$，当$\overline{I}_9$=0（有效）时，$\overline{Y}_3 \sim \overline{Y}_0$=0110（“9”的BCD码的 反码），依此类推。

$\overline{I}_9$	$\overline{I}_8$	$\overline{I}_7$	$\overline{I}_6$	$\overline{I}_5$	$\overline{I}_4$	$\overline{I}_3$	$\overline{I}_2$	$\overline{I}_1$	$\overline{I}_0$	$\overline{Y}_3$	$\overline{Y}_2$	$\overline{Y}_1$	$\overline{Y}_0$
0	x	x	x	x	x	x	x	x	x	0	1	1	0
1	0	x	x	x	x	x	x	x	x	0	1	1	1
1	1	0	x	x	x	x	x	x	x	1	0	0	0
1	1	1	0	x	x	x	x	x	x	1	0	0	1
1	1	1	1	0	x	x	x	x	x	1	0	1	0
1	1	1	1	1	0	x	x	x	x	1	0	1	1
1	1	1	1	1	1	0	x	x	x	1	1	0	0
1	1	1	1	1	1	1	0	x	x	1	1	0	1
1	1	1	1	1	1	1	1	0	x	1	1	1	0
1	1	1	1	1	1	1	1	1	0	1	1	1	1

低电平输入有效

输出等于优先级最高的输入信号对应编号的反码

优先编码器（74147）的Verilog HDL源程序

❖利用if_else语句的分支具有先后顺序的特点，用if_else语句可方便地实现优先编码器。

```
module
 CT74147(IN0,IN1,IN2,IN3,IN4,IN5,
IN6,IN7,IN8,IN9,YN0,YN1,YN2,YN3);
   input    IN0,IN1,IN2,IN3, IN4,
            IN5,IN6,IN7,IN8,IN9;
  output    YN0,YN1,YN2,YN3;
  reg       YN0,YN1,YN2,YN3;
  reg[3:0]  Y; //中间变量
  always @(IN0 or IN1 or IN2 or
           IN3 or IN4 or IN5 or IN6
           or IN7 or IN8 or IN9)
    begin
    if (IN9 == 1'b0)        Y = 4'b0110;
    else if (IN8 == 1'b0)   Y = 4'b0111;
    else if (IN7 == 1'b0)   Y = 4'b1000;
    else if (IN6 == 1'b0)   Y = 4'b1001;
    else if (IN5 == 1'b0)   Y = 4'b1010;
    else if (IN4 == 1'b0)   Y = 4'b1011;
    else if (IN3 == 1'b0)   Y = 4'b1100;
    else if (IN2 == 1'b0)   Y = 4'b1101;
    else if (IN1 == 1'b0)   Y = 4'b1110;
    else if (IN0 == 1'b0)   Y = 4'b1111;
    YN0 = Y[0];
    YN1 = Y[1];
    YN2 = Y[2];
    YN3 = Y[3];
  end
endmodule
```

电平有效型输入在参数表中可以忽略，也可写为always

译码器

❖将二进制代码所表示的信息翻译成对应输出的高低电平信号的过程称为**译码**，译码是编码的反操作。实现译码功能的电路称为**译码器**（Decoder）。

❖常用的译码器有变量译码器、码制变换译码器和显示译码器。

- **变量译码器**（**二进制译码器**）是用来表示输入变量状态全部组合的译码器。n个输入代码有2^n个状态，因此n位二进制译码器有n个输入端和2^n个输出端，一般称为**n线-2^n线译码器**。常用的有双2线-4线译码器74××139，3线-8线译码器74××138，4线-16线译码器74××154等。
- **码制变换译码器**是将输入的某个进制代码转换成对应的其他码制输出的译码器。如二-十进制码（8421码）至十进制码译码器（简称**BCD译码器**）、余3码至十进制码译码器、余3循环码至十进制码译码器等。
- **显示译码器**是将输入代码转换成驱动7段数码显示器各段的电平信号的译码器。常用的有74××47（低电平输出有效）、74××49（高电平输出有效）、74××48（高电平输出有效）等。

二进制译码器

❖**二进制译码器**将每个输入二进制代码译成对应的一根输出线上的高电平（或低电平）信号。因此称为**n线-2^n线译码器**。

❖**二进制编码器**将输入的每一个高（或低）电平信号编成一个对应的二进制代码。为避免输出混乱，任何时刻只允许一个输入信号有效。

❖二进制译码器功能与二进制编码器正好相反，它是将具有特定含义的不同二进制代码辨别出来，并转换成相应的电平信号。

特点

① 任何时刻最多**只允许1个输出有效**

——与编码器对应，任何时刻只允许有一个输入为有效电平

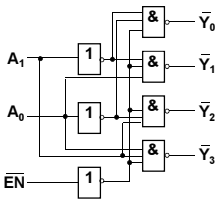
② 高电平输出有效时，每个输出都是对应的输入变量最小项；**低**电平输出有效时，每个输出都是对应的输入变量最小项的**反**

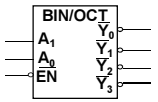
——二进制译码器也称为**最小项译码器**。

2线-4线译码器（74139）

❖**低电平**输出有效

❖$\overline{EN}$为使能控制端（选通信号），为**0**时，译码器处于**工作**状态；为**1**时，处于**禁止**工作状态。





$\overline{Y}_0 = \overline{\overline{A}_1\overline{A}_0} = \overline{m}_0$　　$\overline{Y}_1 = \overline{\overline{A}_1 A_0} = \overline{m}_1$

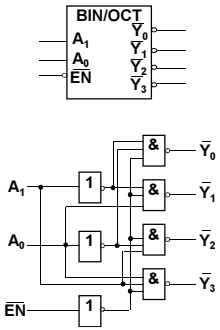
$\overline{Y}_2 = \overline{A_1\overline{A}_0} = \overline{m}_2$　　$\overline{Y}_3 = \overline{A_1 A_0} = \overline{m}_3$

➢当低电平输出有效时，每个输出都是对应的输入变量**最小项**的**反**

功能表

$\overline{EN}$	A_1	A_0	$\overline{Y}_3$	$\overline{Y}_2$	$\overline{Y}_1$	$\overline{Y}_0$
1	X	X	1	1	1	1
0	0	0	1	1	1	0
0	0	1	1	1	0	1
0	1	0	1	0	1	1
0	1	1	0	1	1	1

2线-4线译码器（74139）的手工设计方法



③ 画出逻辑图

① 真值表

$\overline{EN}$	A_1	A_0	$\overline{Y_3}$	$\overline{Y_2}$	$\overline{Y_1}$	$\overline{Y_0}$
1	X	X	1	1	1	1
0	0	0	1	1	1	0
0	0	1	1	1	0	1
0	1	0	1	0	1	1
0	1	1	0	1	1	1

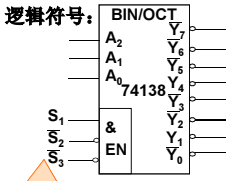
② 根据真值表推导出逻辑表达式（最大项推导法）

$$\overline{Y}_0 = A_1 + A_0 = \overline{\overline{A_1} + \overline{A_0}} = \overline{A_1}\,\overline{A_0} = \overline{m}_0 \qquad \overline{Y}_1 = \overline{A_1}\,\overline{A_0} = \overline{m}_1$$

$$\overline{Y}_2 = \overline{A_1}\,\overline{\overline{A_0}} = \overline{m}_2 \qquad \overline{Y}_3 = \overline{A_1}\,\overline{A_0} = \overline{m}_3$$

3线-8线译码器（74138）

逻辑符号：



S_1、/S_2、/S_3为3个使能输入端，只有当它们分别为1、0、0时，译码器才正常译码；否则禁止工作

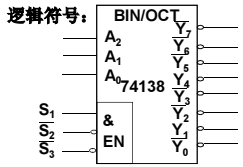
功能表

$S_1\overline{S_2}\overline{S_3}$	A_2 A_1 A_0	$\overline{Y_7}\overline{Y_6}\overline{Y_5}\overline{Y_4}\overline{Y_3}\overline{Y_2}\overline{Y_1}\overline{Y_0}$
≠100	X X X	1 1 1 1 1 1 1 1
=100	0 0 0	1 1 1 1 1 1 1 0
=100	0 0 1	1 1 1 1 1 1 0 1
=100	0 1 0	1 1 1 1 1 0 1 1
=100	0 1 1	1 1 1 1 0 1 1 1
=100	1 0 0	1 1 1 0 1 1 1 1
=100	1 0 1	1 1 0 1 1 1 1 1
=100	1 1 0	1 0 1 1 1 1 1 1
=100	1 1 1	0 1 1 1 1 1 1 1

$$\overline{Y}_0 = \overline{A_2}\,\overline{A_1}\,\overline{A_0} = \overline{m}_0;\ \overline{Y}_1 = \overline{A_2}\,\overline{A_1}A_0 = \overline{m}_1$$
$$\overline{Y}_2 = \overline{A_2}A_1\,\overline{A_0} = \overline{m}_2;\ \overline{Y}_3 = \overline{A_2}A_1A_0 = \overline{m}_3$$
$$\overline{Y}_4 = A_2\,\overline{A_1}\,\overline{A_0} = \overline{m}_4;\ \overline{Y}_5 = A_2\,\overline{A_1}A_0 = \overline{m}_5$$
$$\overline{Y}_6 = A_2A_1\,\overline{A_0} = \overline{m}_6;\ \overline{Y}_7 = A_2A_1A_0 = \overline{m}_7$$

74138的手工设计方法

逻辑符号：



② 根据真值表推导出逻辑表达式（最大项推导法）

③ 画出逻辑图

① 真值表

$S_1\overline{S_2}\overline{S_3}$	A_2 A_1 A_0	$\overline{Y_7}\overline{Y_6}\overline{Y_5}\overline{Y_4}\overline{Y_3}\overline{Y_2}\overline{Y_1}\overline{Y_0}$
≠100	X X X	1 1 1 1 1 1 1 1
=100	0 0 0	1 1 1 1 1 1 1 0
=100	0 0 1	1 1 1 1 1 1 0 1
=100	0 1 0	1 1 1 1 1 0 1 1
=100	0 1 1	1 1 1 1 0 1 1 1
=100	1 0 0	1 1 1 0 1 1 1 1
=100	1 0 1	1 1 0 1 1 1 1 1
=100	1 1 0	1 0 1 1 1 1 1 1
=100	1 1 1	0 1 1 1 1 1 1 1

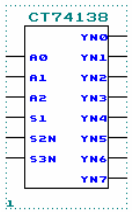
$$\overline{Y}_0 = \overline{A_2}\,\overline{A_1}\,\overline{A_0} = \overline{m}_0;\ \overline{Y}_1 = \overline{A_2}\,\overline{A_1}A_0 = \overline{m}_1$$
$$\overline{Y}_2 = \overline{A_2}A_1\,\overline{A_0} = \overline{m}_2;\ \overline{Y}_3 = \overline{A_2}A_1A_0 = \overline{m}_3$$
$$\overline{Y}_4 = A_2\,\overline{A_1}\,\overline{A_0} = \overline{m}_4;\ \overline{Y}_5 = A_2\,\overline{A_1}A_0 = \overline{m}_5$$
$$\overline{Y}_6 = A_2A_1\,\overline{A_0} = \overline{m}_6;\ \overline{Y}_7 = A_2A_1A_0 = \overline{m}_7$$

74138的Verilog HDL设计方法（1/2）

HDL设计—— 采用if-else语句和case语句描述

分析：分两种情况

- 当输入S1、/S2、/S3为100时，译码器工作；当S1、/S2、/S3不等于100时，译码器禁止工作——适合用if-else语句描述。
- 在译码器工作时，根据输入A2~A0的取值不同，某一个输出信号为有效电平——适合用case语句描述。




```
module CT74138(A0,A1,A2,S1,S2N,S3N,YN0,
        YN1,YN2,YN3,YN4,YN5,YN6,YN7);
  input     A0,A1,A2,S1,S2N,S3N;
  output    YN0,YN1,YN2,YN3,YN4,YN5,YN6,YN7;
  reg       YN0,YN1,YN2,YN3,YN4,YN5,YN6,YN7;
  reg[7:0]  Y_SIGNAL;
```

中间变量，便于对多个信号一次赋值

74138的Verilog HDL设计方法（2/2）

使能信号均有效时译码器处于工作状态

```
always
  begin
    if (S1 && !S2N && !S3N==1)
      begin
        case ({A2,A1,A0})
          3 'b000 : Y_SIGNAL =8 'b11111110;
          3 'b001 : Y_SIGNAL =8 'b11111101;
          3 'b010 : Y_SIGNAL =8 'b11111011;
          3 'b011 : Y_SIGNAL =8 'b11110111;
          3 'b100 : Y_SIGNAL =8 'b11101111;
          3 'b101 : Y_SIGNAL = 8'b11011111;
          3 'b110 : Y_SIGNAL =8 'b10111111;
          3 'b111 : Y_SIGNAL =8 'b01111111;
          default : Y_SIGNAL =8'b11111111;
        endcase
      end
```

使能信号无效时译码器处于禁止状态

```
    else Y_SIGNAL = 8'b11111111;
    YN0 = Y_SIGNAL[0];
    YN1 = Y_SIGNAL[1];
    YN2 = Y_SIGNAL[2];
    YN3 = Y_SIGNAL[3];
    YN4 = Y_SIGNAL[4];
    YN5 = Y_SIGNAL[5];
    YN6 = Y_SIGNAL[6];
    YN7 = Y_SIGNAL[7];
  end
endmodule
```


码制变换译码器

❖**码制变换译码器**是将输入的二进制代码转换成对应的其他码制输出的译码器。

- 二-十进制译码器（BCD译码器，4-10线译码器）：将输入的4位8421码翻译成0~9十个十进制数的电路。用于驱动十进制数字显示管、指示灯、继电器
  - 完全译码的BCD译码器：当输入ABCD出现伪码0101~1111时，译码器输出Y_0~Y_9均为“1”，如74xx42（输出为低电平有效）
  - 不完全译码的BCD译码器：当ABCD＝0101~1111时，Y_0~Y_9均为任意值
- 余3码至十进制码译码器
- 余3循环码至十进制码译码器

BCD译码器（74xx42）

- 输出Y_9~Y_0为低电平有效
- 当输入DCBA为无用码（伪码）1010~1111时，译码器输出Y_9~Y_0均为“1”，不会出现低电平，不会产生错误译码

最高位

	D C B A	Y9 Y8 Y7 Y6 Y5 Y4 Y3 Y2 Y1 Y0
0	0 0 0 0	1 1 1 1 1 1 1 1 1 0
1	0 0 0 1	1 1 1 1 1 1 1 1 0 1
2	0 0 1 0	1 1 1 1 1 1 1 0 1 1
3	0 0 1 1	1 1 1 1 1 1 0 1 1 1
4	0 1 0 0	1 1 1 1 1 0 1 1 1 1
5	0 1 0 1	1 1 1 1 0 1 1 1 1 1
6	0 1 1 0	1 1 1 0 1 1 1 1 1 1
7	0 1 1 1	1 1 0 1 1 1 1 1 1 1
8	1 0 0 0	1 0 1 1 1 1 1 1 1 1
9	1 0 0 1	0 1 1 1 1 1 1 1 1 1
不用	1 0 1 0	1 1 1 1 1 1 1 1 1 1
	1 0 1 1	1 1 1 1 1 1 1 1 1 1
	1 1 0 0	1 1 1 1 1 1 1 1 1 1
	1 1 0 1	1 1 1 1 1 1 1 1 1 1
	1 1 1 0	1 1 1 1 1 1 1 1 1 1
	1 1 1 1	1 1 1 1 1 1 1 1 1 1

74xx42的Verilog HDL源程序（1/2）

设计：根据真值表推导出逻辑表达式（最大项推导法），画出逻辑图

也可以直接用HDL来描述功能（case语句或assign语句）

$$Y_0=A+B+C+D=\overline{\bar{A}\bar{B}\bar{C}\bar{D}}$$

$$Y_1=\overline{A\bar{B}\bar{C}\bar{D}}$$

⋮

$$Y_9=\overline{A\bar{B}\bar{C}D}$$

```
module CT7442(D,C,B,A, YN0, YN1,YN2, YN3,YN4,
        YN5,YN6,YN7,YN8,YN9);
  input      D,C,B,A;
  output     YN0,YN1,YN2,YN3,YN4,YN5, YN6,YN7,YN8,YN9;
  reg        YN0,YN1,YN2,YN3,YN4,YN5, YN6,YN7,YN8,YN9;
  reg[9:0]   Y_SIGNAL;
```

中间变量，便于对多个信号一次赋值

74xx42的Verilog HDL源程序（2/2）

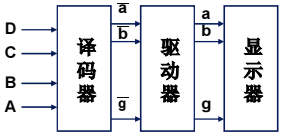
```
always
  begin
    case ({D,C,B,A})
      'b0000 : Y_SIGNAL = 'b1111111110;
      'b0001 : Y_SIGNAL = 'b1111111101;
      'b0010 : Y_SIGNAL = 'b1111111011;
      'b0011 : Y_SIGNAL = 'b1111110111;
      'b0100 : Y_SIGNAL = 'b1111101111;
      'b0101 : Y_SIGNAL = 'b1111011111;
      'b0110 : Y_SIGNAL = 'b1110111111;
      'b0111 : Y_SIGNAL = 'b1101111111;
      'b1000 : Y_SIGNAL = 'b1011111111;
      'b1001 : Y_SIGNAL = 'b0111111111;
      default :Y_SIGNAL = 'b1111111111;
    endcase
```

```
    YN0 = Y_SIGNAL[0];
    YN1 = Y_SIGNAL[1];
    YN2 = Y_SIGNAL[2];
    YN3 = Y_SIGNAL[3];
    YN4 = Y_SIGNAL[4];
    YN5 = Y_SIGNAL[5];
    YN6 = Y_SIGNAL[6];
    YN7 = Y_SIGNAL[7];
    YN8 = Y_SIGNAL[8];
    YN9 = Y_SIGNAL[9];
  end
endmodule
```


显示译码器

❖ 显示译码器用于驱动数码显示器，是一种将二进制代码表示的数字、文字、符号用人们习惯的形式直观显示出来的电路。

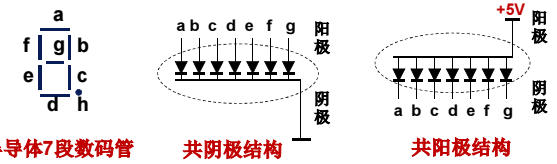
（1）电路结构（8421BCD译码显示电路）



① 显示器件

- 辉光数码管、7段荧光数码管、液晶显示器
- 目前广泛使用的显示数字的器件是7段数码显示器（由7段可发光的线段拼合而成），包括半导体数码显示器和液晶显示器两种。
- 数码显示器人们通常又称为数码管。半导体7段数码管实际上是由7个发光二极管（LED）构成的，因此又称为LED数码管。

半导体7段数码管



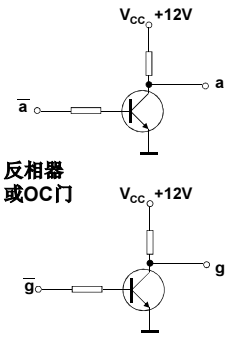
半导体7段数码管　　共阴极结构　　共阳极结构

❖ 若使用共阴极LED数码管，则显示译码器的输出应为高电平输出有效；若使用共阳极LED数码管，则译码器应为低电平输出有效。

驱动电路和译码器

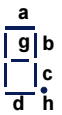
② 驱动电路

使译码器输出反相，并提供大的驱动电流。



③ 译码器（共阳器件）

DCBA	$\overline{a}$ $\overline{b}$ $\overline{c}$ $\overline{d}$ $\overline{e}$ $\overline{f}$ $\overline{g}$	显示数字
0000	0 0 0 0 0 0 1	0
0001	1 0 0 1 1 1 1	1
0010	0 0 1 0 0 1 0	2
0011	0 0 0 0 1 1 0	3
0100	1 0 0 1 1 0 0	4
0101	0 1 0 0 1 0 0	5
0110	1 1 0 0 0 0 0	6
0111	0 0 0 1 1 1 1	7
1000	0 0 0 0 0 0 0	8
1001	0 0 0 1 1 0 0	9
1010	X X X X X X X	
1011	X X X X X X X	
1100	X X X X X X X	
1101	X X X X X X X	
1110	X X X X X X X	
1111	X X X X X X X	

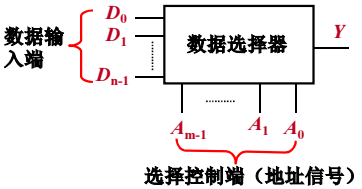


第二部分：组合逻辑

一. 逻辑门电路
二. 布尔代数及其门电路实现
三. Verilog HDL介绍
1. Verilog HDL基本结构
2. Verilog HDL的词法
3. Verilog HDL常用语句
4. 不同抽象级别的Verilog HDL模型
四. 基本组合逻辑部件设计
1. 运算单元电路
2. 编码器/译码器
3. 数据选择器

数据选择器

- 从一组输入数据选出其中需要的一个数据作为输出的过程叫做**数据选择**，具有数据选择功能的电路称为**数据选择器**（Data Selector）。
- 数据选择器又称**多路选择器**（Multiplexer，多路器），它是以“与或非”门或以“与或”门为主体的组合逻辑电路。它在选择控制信号的作用下，能从多路平行输入数据中任选一路数据作为输出。
- 常用的集成数据选择器有四2选1（74××157）、双4选1（74××153）、8选1（74××151）及16选1（74××150）数据选择器等。

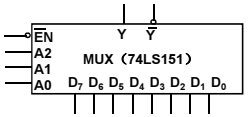


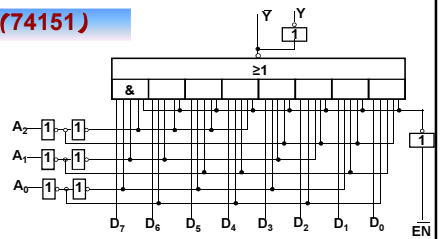
若有n个输入，则称为n选1数据选择器。

8选1数据选择器（74151）

2、8选1数据选择器（74151）

❖逻辑图与逻辑符号





$$Y = \overline{A_2}\,\overline{A_1}\,\overline{A_0}D_0 + \overline{A_2}\,\overline{A_1}A_0D_1 + \overline{A_2}A_1\overline{A_0}D_2 + \overline{A_2}A_1A_0D_3 + A_2\overline{A_1}\,\overline{A_0}D_4 + A_2\overline{A_1}A_0D_5 + A_2A_1\overline{A_0}D_6 + A_2A_1A_0D_7 \quad (\overline{EN} = 0)$$
$$= m_0D_0 + m_1D_1 + m_2D_2 + m_3D_3 + m_4D_4 + m_5D_5 + m_6D_6 + m_7D_7$$

❖功能

①$A_2A_1A_0$为控制端——

8选1数据选择器（8路开关）

在/EN=0时

8选1数据选择器的功能（2）

功能表（$\overline{EN}$=0）

A_2	A_1	A_0	Y
0	0	0	D_0
0	0	1	D_1
0	1	0	D_2
0	1	1	D_3
1	0	0	D_4
1	0	1	D_5
1	1	0	D_6
1	1	1	D_7

功能②：$D_7 \sim D_0$为控制端——多功能运算电路

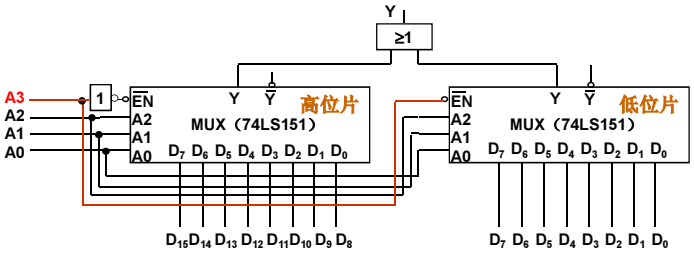
- 通过D_7~D_0取不同的值，从输入变量A_2、A_1、A_0的各个最小项中选取某几个最小项输出，实现不同的运算电路
- 有2^8=256种功能——包含3变量的各种最小项表达式——可实现任意组合逻辑电路的设计。

$$Y = D_0m_0 + D_1m_1 + D_2m_2 + D_3m_3 + D_4m_4 + D_5m_5 + D_6m_6 + D_7m_7$$

当$D_7 \sim D_0$为0000_0000时，Y=0；
当$D_7 \sim D_0$为1111_1111时，Y=1；
当$D_7 \sim D_0$为0000_0001时，Y=m_0；
当$D_7 \sim D_0$为1010_0101时，Y=m_7+m_5+m_2+m_0。

数据选择器（74151）的扩展

应用使能端将2片74151（8选1）扩展为16选1数据选择器

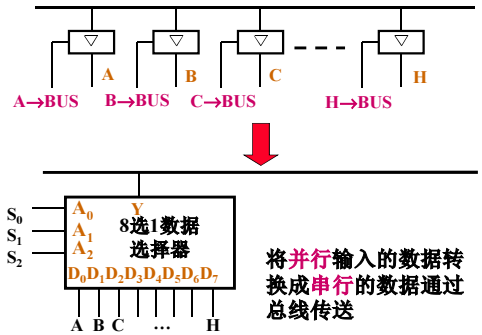


- A_3=0时，低位片工作，从输入D_7~D_0中选择1个输出；
- A_3=1时，高位片工作，从输入D_{15}~D_8中选择1个输出。

数据选择器的应用（1/2）

❖数据选择器除选择功能外，还有其他的用途

- 代替三态门，实现总线发送控制（并串转换）



将并行输入的数据转换成串行的数据通过总线传送

$S_2S_1S_0$	Y
000	A
001	B
010	C
011	D
100	E
101	F
110	G
111	H

数据选择器的应用（2/2）

❖函数发生器——用数据选择器实现任意组合逻辑函数

数据选择器可以看成是用N个控制端D_{N-1}~D_0选择2^N个最小项的某几个组成“与-或”表达式。选择某些控制信号D_i为“1”，就是选中这些最小项组成逻辑函数。

❖如果给定一个组合逻辑函数，如何用数据选择器实现？

- 利用互补律，将组合逻辑函数变换为最小项之和的标准形式；
- 根据该逻辑函数有几个输入变量，确定数据选择器的地址输入为几位，将逻辑函数的输入变量作为数据选择器的地址输入；
- 写出数据选择器的输出表达式；
- 比较逻辑函数的标准形式与数据选择器的输出表达式，推导出数据选择器的D_i哪些为1，哪些为0；
- 画出电路图。

利用数据选择器实现逻辑函数

利用数据选择器实现逻辑函数

$$F(A,B,C)=\overline{A}\overline{B}C+\overline{A}B\overline{C}+AB$$

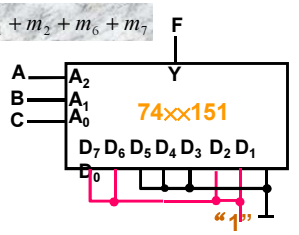
解：使用8选1数据选择器74151

将逻辑函数的输入变量作为数据选择器的地址输入。

首先将组合逻辑函数变换为最小项之和的标准形式：

$$F=\overline{A}\overline{B}C+\overline{A}B\overline{C}+AB(\overline{C}+C)=m_1+m_2+m_6+m_7$$

74151 输出 $Y=D_0m_0+D_1m_1+D_2m_2+D_3m_3+D_4m_4+D_5m_5+D_6m_6+D_7m_7$



比较*F*和*Y*，得：

D_0=0，D_1=1，D_2=1，D_3=0，

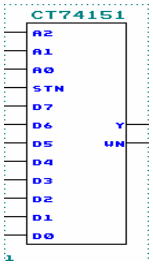
D_4=0，D_5=0，D_6=1，D_7=1

数据选择器（74151）的HDL设计

❖信号定义

- D7～D0：8位数据输入端
- A2～A0：地址输入端
- STN：使能控制端（低电平有效）
- Y：同相数据输出端
- WN：反相数据输出端，即WN是Y的反相输出

74151的元件符号：



74151的Verilog HDL源程序

❖HDL设计

采用if-else语句和case语句描述

```
module CT74151(A2,A1,A0,STN,D7,
    D6,D5,D4,D3,D2,D1,D0,Y,WN);
  input     A2,A1,A0,STN;
  input     D7,D6,D5,D4,D3,
            D2,D1,D0;
  output    Y,WN;
  reg       Y,WN;
  always
    begin
```

```
      if (STN == 0)
        begin
          case ({A2,A1,A0})
            3'b000 : Y = D0;
            3'b001 : Y = D1;
            3'b010 : Y = D2;
            3'b011 : Y = D3;
            3'b100 : Y = D4;
            3'b101 : Y = D5;
            3'b110 : Y = D6;
            3'b111 : Y = D7;
          endcase
        end
      else Y = 1'b0;
      WN = ~Y;
    end
endmodule
```